

全球资深Python专家亲自执笔，Python语言的核心开发人员作序鼎力推荐

以案例为导向，全面讲解标准库中百余模块的使用方法和工作原理，简洁易懂，可操作性极强

The Python Standard Library by Example

Python标准库

（美） Doug Hellmann 著

刘炽 等译



机械工业出版社
China Machine Press

通过大量精心挑选的示例掌握强大的Python标准库!

Python标准库包含数百个模块,可以用来与操作系统、解释器和Internet交互——所有这些模块都已经过充分测试,可以直接在日常开发中使用。本书精心设计了大量示例,以方便读者学习和使用标准库。

本书作者拥有12年以上Python开发经验,是Python软件基金会的信息交流主管,他的“Python Module of the Week”系列博客文章享誉整个Python社区,本书便是以这一系列博客文章为基础,以示例的方式展示了标准库中的每个模块是如何工作的以及为什么要这样工作。

在本书中,你会看到用来处理文本、数据类型、算法、数学计算、文件系统、网络通信、Internet、XML、Email、加密、并发性、运行时和语言服务等各个方面的实用代码和解决方案。在内容安排上,每一节都会全面介绍一个模块,并提供一些很有价值的补充资源链接,这使得本书成为一本理想的Python标准库参考手册。

本书涵盖以下内容:

- 用string、textwrap、re和difflib处理文本;
- 实现数据结构: collections、array、queue、struct、copy, 等等;
- 读、写和管理文件及目录;
- 正则表达式模式匹配;
- 交换数据和提供持久存储;
- 归档和数据压缩;
- 管理进程和线程;
- 使用应用“构建模块”: 解析命令行选项、提示输入密码、调度事件和日志记录;
- 测试、调试和编译;
- 控制运行时配置;
- 使用模块和包工具。

如果你刚接触Python,本书将带你迅速进入一个全新的世界。如果你以前用过Python,你会发现一些新的强大的解决方案,对于你之前尝试过的模块,你会发现更好的用法。

客服热线:(010) 88378991, 88361066
购书热线:(010) 68326294, 88379649, 68995259
投稿热线:(010) 88379604
读者信箱:hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

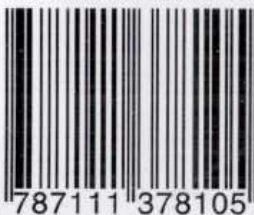
网上购书: www.china-pub.com

PEARSON

www.pearson.com

上架指导: 计算机/程序设计/Python

ISBN 978-7-111-37810-5



9 787111 378105

定价: 139.00元

译者序

Python 的设计哲学是“优雅”、“明确”、“简单”。因此，Python 开发者的哲学是“用一种方法，最好是只有一种方法来做一件事”。不仅如此，Python 设计为一种可扩展的语言，并非所有的特性和功能都集成到语言核心。随 Python 附带安装有 Python 标准库，标准库包含大量极其有用的模块。熟悉 Python 标准库十分重要，因为如果熟悉这些库中的模块，那么大多数问题都可以简单快捷地使用它们来解决。

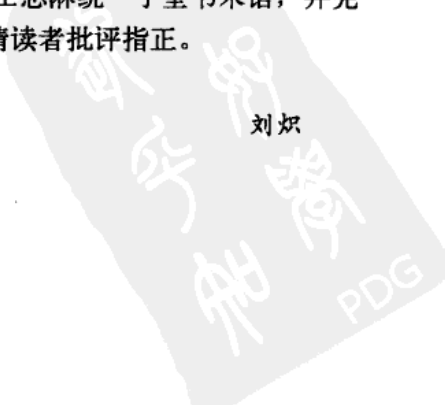
标准库包含数百个模块，为常见任务提供了丰富的工具，可以用来作为应用开发的起点。也许有人发现一个模块正是他想要的，但是不知道如何来使用；也许有人为他的任务挑选了不合适的模块，也可能已经厌倦了一切从头开始。大多时候，一个简短的例子要比一份手册文档更有帮助。这正是本书的出发点。本书会提供一些精选的例子，展示如何使用这些模块中最常用的一些特性。相信你在使用 Python 时遇到的问题都能在本书中得到解答。

本书作者从 1.4 版开始就一直在从事 Python 编程方面的工作，他正是享有盛誉的博客系列“Python Module of the Week”的博主。在这个博客中，他全面研究了标准库的众多模块，利用实际例子来介绍各个模块如何工作。相信很多人已经从他的博客中受益。为满足人们的迫切需求，他把这些博客文章进一步整理完善，形成了你手上的这本书。

本书沿袭了博客的叙事风格，Doug Hellmann 通过轻松的方式，让你从具体的例子、具体的实践中了解技术细节，在知道“怎样做”的同时还能理解“为什么这样做”。相信你已经迫不及待地想要翻开下一页了，那么，进入 Doug 的 Python 世界吧！

本书由刘炽、苏金国、李璜、杨健康等主译，乔会东、仝磊、王少轩、程芳、宋旭民、黄小钰等分别对全书各章进行审阅，另外姚曜、程龙、吴忠望、张练达、陈峰、江健、姚勇、卢鋈、张莹参与了全书的修改整理，林琪、刘亮、刘跃邦、高强和王志淋统一了全书术语，并完善了关键部分的翻译。由于水平有限，译文肯定有不当之处，敬请读者批评指正。

刘炽



序

今天是 2010 年的感恩节。不论人们是否身在美国，在这个节日里，大家都一边品尝丰盛的食物，一边欣赏橄榄球比赛，有些人可能会出门逛逛。

对我（以及其他很多人）来说，会借此机会回顾一下过去的岁月，想想那些让我们的生活充满色彩的人和事，向他们致以感谢。当然，我们每天都该这么做，不过专门有一天来表达谢意有时会让我们想得更深远一些。

现在我坐在这里为本书写序，非常感谢能有机会做这件事，不过我想到的不只是本书的内容，也不只是作者本人（一个无比热情的社区成员），我所想的是这个主题本身——Python，具体来讲，还有它的标准库。

当前发布的每版 Python 都包含数百个模块，它们是多年来多位开发人员针对多个主题和多个任务共同完成的。这些模块涵盖一切，从发送和接收 Email，到 GUI 开发，再到内置的 HTTP 服务器都一应俱全。就其本身而言，标准库的开发和维护是一项极其庞大的工作。如果没有多年来一直维护它的人们，没有数以千计的人提交补丁、文档和提供反馈，它绝不会成为今天的模样。

这是一个惊人的成就，在 Python 日益普及的今天（不论是作为语言还是一种生态系统），标准库已经成为其中不可或缺的重要组成部分。如果没有标准库，没有核心团队和其他人员的“内含动力”（batteries included）口号，Python 绝对不可能走这么远。它已经被成千上万的人和公司下载，并已安装在数百万服务器、台式机和其他设备上。

即使没有标准库，Python 仍是一种很不错的语言，在教学、学习和可读性方面有扎实的基础。基于这些优点，它本身也能发展得足够好。不过标准库把它从一种有趣的体验变成为一个强大而有效的工具。

每一天，全世界的开发人员都在构建工具和完整的应用，他们所基于的只是核心语言和标准库。你不仅要能够明确概念，描述汽车是什么（语言），还要得到足够的部件和工具来自行组装一辆基本的汽车。它可能并不完善，不过可以使你从无到有，这将是一个很好的奖励，会赋予你巨大的动力。我曾反复对那些骄傲地看着我的人说：“看看我构建的应用，除了 Python 提供的，其他的工具通通都没有用到！”

不过，标准库也不是完美无缺的，它也有自己的缺陷。由于标准库的规模、广度和它的“年龄”，毫无疑问有些模块有不同层次的质量、API 简洁性和覆盖性。有些模块存在“特性蔓延”的问题，或者无法跟进其覆盖领域中的最新进展。通过众多不计酬劳的志愿者的帮助和辛勤工作，Python 还在继续发展、壮大和改进。

不过，有些人对 Python 还有争议，不仅由于它的缺点，而且因为标准库并不一定构成其模块涵盖领域中“最顶尖的”解决方案（毕竟，“最佳”是一个不断改变和调整的目

标), 因而认为应当将它完全舍弃, 尽管它还在不断改善。这些人遗漏了一个事实: 标准库不仅是促使 Python 不断成功的一个重要组成部分, 而且尽管存在瑕疵, 它还是一个绝妙的资源。

不过我有意忽略了一个巨大的领域: 文档。标准库的文档很不错, 还在继续改进和发展。由于标准库的庞大规模和广度, 相应的文档规模也很惊人。在数以千计的开发人员和用户的努力下, 我们有成百上千页文档, 实在让人佩服。每天都有数万人在使用这些文档创建应用——可能简单到只是一页的脚本, 也可能很复杂, 比如控制大型机械手的软件。

正是由于文档, 我们才会看到本书。所有好的文档和代码都有一个起点——关于目标“是什么”以及“将是什么”要有一个核心概念。从这个内核出发, 才有了角色 (API) 和故事情节 (模块)。谈到代码, 有时代码会从一个简单的想法开始: “我想解析一个字符串, 查找一个日期。”不过等结束时, 你可能已经查看了数百个单元测试、函数以及你编写的其他代码, 你会坐下来, 发现自己构建的东西远远超出了原先的设想。文档也是如此, 特别是代码文档。

在我看来, 代码文档中最重要的部分就属于这种例子。要写有关一个 API 中某一部分的描述, 你可能会写上几本书, 可以用华丽的文字和经过深思熟虑的用例来描述松耦合的接口。不过, 如果第一次查看这个描述的用户无法将这些华丽的文字、仔细考量的用例和 API 签名结合在一起, 构建出有意义的应用并解决他们的问题, 这一切就完全是徒劳。

人们建立重要连接所用的网关也属于这种例子, 这些逻辑会从一个抽象概念跳转到具体的事物。“了解”思想和 API 是一回事; 知道它如何使用则是另外一回事。如果你不仅想要学到东西, 还希望改善现状, 这会很有帮助。

这就把我们重新引回 Python。本书作者 Doug Hellmann 在 2007 年创建了一个名为 “Python Module of the Week” 的博客。在这个博客中, 他全面研究了标准库的众多模块, 采用一种“示例为先”的方式介绍各个模块如何工作以及为什么。从读到它的第一天起, 它就成为我除了核心 Python 文档之外的又一个必访之地。他的作品已经成为我以及 Python 社区其他人不可缺少的必备资源。

Doug 的文章填补了当前我所看到的 Python 文档的一个重大空白: 对例子的迫切需求。用一种有效而简单的方式展示如何做以及为什么这么做, 这绝非易事。我们已经看到, 这也是一项很重要、很有价值的工作, 对人们每一天的工作都有帮助。人们频繁地给我发邮件, 告诉我: “你看过 Doug 的这个帖子吗? 实在太棒了!” 或者 “为什么这个不能放在核心文档里呢? 它能帮助我了解到底是怎么做的!”

当我听说 Doug 准备花些时间进一步完善他现有的工作, 把它变成一本书, 让我能把它放在桌子上反复翻阅, 以备急用, 我真是太兴奋了。Doug 是一位非凡的技术作者, 而且能敏锐地捕捉到细节。有一整本书专门讲解实际例子, 介绍标准库中一百多个模块是如何工作的, 而且是他来写这本书, 实在让我欣喜若狂。

所以, 我要感谢 Python, 感谢标准库 (包括它的瑕疵), 感谢我拥有的这个活力充沛有时也存在问题的庞大 Python 社区。我要感谢核心开发小组的辛勤工作, 包括过去、现在, 还有将

来。我还要感谢这么多社区成员提供的资源、投入的时间和做出的努力，其中 Doug Hellmann 更是卓越的代表，正是这些让这个社区和生态系统如此生机勃勃。

最后，我要感谢本书。继续向它的作者表示敬意，这本书会在未来几年得到充分利用。

Jesse Noller

Python 核心开发人员

PSF Board 成员

Nasuni 公司首席工程师



前言

随每个 Python 版本的发布会同时发布标准库，标准库包含数百个模块，为操作系统、解释器和互联网之间的交互提供了丰富的工具。所有这些模块都得到充分测试，可以用来作为应用开发的起点。本书会提供一些精选的例子，展示如何使用这些模块中最常用的一些特性，正是这些特性使 Python 有了“内含动力”的口号。这些例子均取自流行的“Python Module of the Week (PyMOTW)”博客系列。

本书读者对象

本书的读者应该是中等水平的 Python 程序员，所以尽管书中对所有源代码都做了讨论，但只有一部分会逐行给出解释。每节会通过源代码和完全独立的示例程序的输出来重点介绍一个模块的特性。我会尽可能简洁地介绍各个特性，使读者能够把重点放在所展示的模块或函数上，而不会因支持代码而分心。

熟悉其他语言的有经验的程序员可以从本书了解 Python，不过本书并不是这种语言的入门读物。研究这些例子时，如果之前对编写 Python 程序有些经验会很有帮助。

有些章节（比如 6.7.9 节和 9.2 节）还需要一些领域特定的知识。这里会提供解释这些例子所需的基本信息，不过由于标准库中模块涵盖的主题如此宽泛，因此不可能在一本书中全面地介绍每一个主题。在每个模块的讨论之后，还提供一个推荐资源列表，可以进一步阅读这些资源，从中了解更多信息。其中包括在线资源、RFC 标准文档以及相关图书。

尽管目前向 Python 3 的过渡正在进行当中，Python 2 仍可能是这几年生产环境中使用的主要 Python 版本，这是因为存在大量遗留 Python 2 源代码，另外向 Python 3 过渡的速度相当迟缓。本书中所有例子的源代码都由原来的在线版本做了更新，并用 Python 2.7（这是 Python 2.x 系列的最后一个版本）进行了测试。很多示例程序完全可以在 Python 3 下工作，不过有些例子涉及的模块已经改名或者已经废弃。

本书组织结构

模块均分组在不同章中介绍，以便查找单个模块来加以引用，并且可以按主题浏览做更深层次的探讨。相对于 <http://docs.python.org> 上全面的参考指南，本书可以作为补充，提供完全可用的示例程序来展示参考指南中介绍的特性。

下载示例代码

原来的博客文章、本书勘误以及示例代码都可以从作者的网站 (<http://www.doughellmann.com/books/byexample>) 下载。

致谢

如果没有大家的贡献和支持，本书绝无可能出现。

1997 年 Dick Wall 让我第一次接触到 Python，那时我们正在 ERDAS 一起合作开发 GIS 软件。我记得，当我发现这样一门如此简便易用的新工具语言时，立刻就喜欢上了它，还对公司不让我们用它来完成“实际工作”颇有不满。在接下来的所有工作中我都大量使用了 Python，我要感谢 Dick，正是因为他，给我以后的软件开发带来许多快乐时光。

Python 核心开发小组创建了一个由语言、工具和库共同构建的健壮的生态系统，这些库在日益普及，还在不断发现新的应用领域。如果没有他们付出的宝贵时间，没有他们提供的丰富资源，我们都还得花时间一次又一次地一切从头开始。

正如前言中所说的，本书中的材料最初是一系列博客文章。每篇文章都得到了 Python 社区成员的审阅和评论，有纠正，有建议，也有问题，这些评论促使我做出修改，这才有了读者手上这本书。感谢大家每周都花时间来阅读我的博客，谢谢大家的关注。

本书的技术审校人员——Matt Culbreth、Katie Cunningham、Jeff McNeil 和 Keyton Weissinger——花了大量时间查找示例代码和相关解释中存在的问题。最终的作品远比我靠一人之力得到的结果强得多。我还得到了 Jesse Noller 对于 multiprocessing 模块以及 Brett Cannon 关于创建定制导入工具提出的很多建议。

还要特别感谢 Pearson 的编辑和制作人员，感谢大家辛苦的工作和一贯的支持，帮助我明确本书的目标。

最后，我要感谢我的妻子 Theresa Flynn，她总是能提出最棒的写作建议，在完成本书的整个过程中，她都一如既往地鼓励我、支持我。当她告诉我，“要知道，某些情况下你要坐下来把它写出来”，我真的很佩服她能如此洞察一切。下面轮到你了。



目 录

译者序		
序		
前言		
第 1 章 文本	1	
1.1 string——文本常量和模板	1	
1.1.1 函数	1	
1.1.2 模板	2	
1.1.3 高级模板	4	
1.2 textwrap——格式化文本段落	6	
1.2.1 示例数据	6	
1.2.2 填充段落	6	
1.2.3 去除现有缩进	7	
1.2.4 结合 dedent 和 fill	7	
1.2.5 悬挂缩进	8	
1.3 re——正则表达式	9	
1.3.1 查找文本中的模式	9	
1.3.2 编译表达式	10	
1.3.3 多重匹配	11	
1.3.4 模式语法	12	
1.3.5 限制搜索	22	
1.3.6 用组解析匹配	23	
1.3.7 搜索选项	28	
1.3.8 前向或后向	36	
1.3.9 自引用表达式	40	
1.3.10 用模式修改字符串	44	
1.3.11 利用模式拆分	46	
1.4 difflib——比较序列	49	
1.4.1 比较文本体	49	
1.4.2 无用数据	51	
1.4.3 比较任意类型	53	
第 2 章 数据结构	55	
2.1 collections——容器数据类型	56	
2.1.1 Counter	56	
2.1.2 defaultdict	59	
2.1.3 deque	59	
2.1.4 namedtuple	63	
2.1.5 OrderedDict	65	
2.2 array——固定类型数据序列	66	
2.2.1 初始化	67	
2.2.2 处理数组	67	
2.2.3 数组与文件	68	
2.2.4 候选字节顺序	68	
2.3 heapq——堆排序算法	69	
2.3.1 示例数据	70	
2.3.2 创建堆	70	
2.3.3 访问堆的内容	72	
2.3.4 堆的数据极值	73	
2.4 bisect——维护有序列表	74	
2.4.1 有序插入	74	
2.4.2 处理重复	75	
2.5 Queue——线程安全的 FIFO 实现	76	
2.5.1 基本 FIFO 队列	77	
2.5.2 LIFO 队列	77	
2.5.3 优先队列	78	

2.5.4 构建一个多线程播客客户程序	79	3.2.4 过滤	119
2.6 struct——二进制数据结构	81	3.2.5 数据分组	121
2.6.1 函数与 Struct 类	81	3.3 operator——内置操作符的函数	
2.6.2 打包和解包	81	接口	123
2.6.3 字节序	82	3.3.1 逻辑操作	123
2.6.4 缓冲区	84	3.3.2 比较操作符	124
2.7 weakref——对象的非永久引用	85	3.3.3 算术操作符	124
2.7.1 引用	85	3.3.4 序列操作符	126
2.7.2 引用回调	86	3.3.5 原地操作符	127
2.7.3 代理	87	3.3.6 属性和元素“获取方法”	128
2.7.4 循环引用	87	3.3.7 结合操作符和定制类	129
2.7.5 缓存对象	92	3.3.8 类型检查	130
2.8 copy——复制对象	94	3.4 contextlib——上下文管理器工具	131
2.8.1 浅副本	94	3.4.1 上下文管理器 API	131
2.8.2 深副本	95	3.4.2 从生成器到上下文管理器	134
2.8.3 定制复制行为	96	3.4.3 嵌套上下文	135
2.8.4 深副本中的递归	96	3.4.4 关闭打开的句柄	136
2.9 pprint——美观打印数据结构	98	第 4 章 日期和时间	138
2.9.1 打印	99	4.1 time——时钟时间	138
2.9.2 格式化	99	4.1.1 壁挂钟时间	138
2.9.3 任意类	100	4.1.2 处理器时钟时间	139
2.9.4 递归	101	4.1.3 时间组成	140
2.9.5 限制嵌套输出	101	4.1.4 处理时区	141
2.9.6 控制输出宽度	101	4.1.5 解析和格式化时间	143
第 3 章 算法	103	4.2 datetime——日期和时间值管理	144
3.1 functools——管理函数的工具	103	4.2.1 时间	144
3.1.1 修饰符	103	4.2.2 日期	145
3.1.2 比较	111	4.2.3 timedelta	147
3.2 itertools——迭代器函数	114	4.2.4 日期算术运算	148
3.2.1 合并和分解迭代器	114	4.2.5 比较值	149
3.2.2 转换输入	116	4.2.6 结合日期和时间	150
3.2.3 生成新值	117	4.2.7 格式化和解析	151

4.2.8 时区	151	5.4.6 常用计算	184
4.3 calendar——处理日期	152	5.4.7 指数和对数	186
4.3.1 格式化示例	152	5.4.8 角	190
4.3.2 计算日期	155	5.4.9 三角函数	191
第 5 章 数学计算	157	5.4.10 双曲函数	194
5.1 decimal——定点数和浮点数的数学 运算	157	5.4.11 特殊函数	195
5.1.1 Decimal	157	第 6 章 文件系统	197
5.1.2 算术运算	158	6.1 os.path——平台独立的文件名管理	198
5.1.3 特殊值	160	6.1.1 解析路径	198
5.1.4 上下文	160	6.1.2 建立路径	200
5.2 fractions——有理数	165	6.1.3 规范化路径	201
5.2.1 创建 Fraction 实例	165	6.1.4 文件时间	202
5.2.2 算术运算	167	6.1.5 测试文件	203
5.2.3 近似值	168	6.1.6 遍历一个目录树	204
5.3 random——伪随机数生成器	168	6.2 glob——文件名模式匹配	205
5.3.1 生成随机数	168	6.2.1 示例数据	205
5.3.2 指定种子	169	6.2.2 通配符	206
5.3.3 保存状态	170	6.2.3 单字符通配符	207
5.3.4 随机整数	171	6.2.4 字符区间	207
5.3.5 选择随机元素	172	6.3 linecache——高效读取文本文件	208
5.3.6 排列	172	6.3.1 测试数据	208
5.3.7 采样	174	6.3.2 读取特定行	209
5.3.8 多个并发生成器	175	6.3.3 处理空行	209
5.3.9 SystemRandom	176	6.3.4 错误处理	210
5.3.10 非均匀分布	177	6.3.5 读取 Python 源文件	210
5.4 math——数学函数	178	6.4 tempfile——临时文件系统对象	211
5.4.1 特殊常量	178	6.4.1 临时文件	211
5.4.2 测试异常值	179	6.4.2 命名文件	213
5.4.3 转换为整数	180	6.4.3 临时目录	214
5.4.4 其他表示	181	6.4.4 预测名	214
5.4.5 正号和负号	183	6.4.5 临时文件位置	215
		6.5 shutil——高级文件操作	216

6.5.1 复制文件	216	第 7 章 数据持久存储与交换	267
6.5.2 复制文件元数据	218	7.1 pickle——对象串行化	268
6.5.3 处理目录树	220	7.1.1 导入	268
6.6 mmap——内存映射文件	222	7.1.2 编码和解码字符串数据	268
6.6.1 读文件	223	7.1.3 处理流	269
6.6.2 写文件	223	7.1.4 重构对象的问题	271
6.6.3 正则表达式	225	7.1.5 不可 pickle 的对象	272
6.7 codecs——字符串编码和解码	226	7.1.6 循环引用	273
6.7.1 Unicode 入门	226	7.2 shelve——对象持久存储	275
6.7.2 处理文件	228	7.2.1 创建一个新 shelf	275
6.7.3 字节序	230	7.2.2 写回	276
6.7.4 错误处理	232	7.2.3 特定 shelf 类型	277
6.7.5 标准输入和输出流	235	7.3 anydbm——DBM 数据库	278
6.7.6 编码转换	238	7.3.1 数据库类型	278
6.7.7 非 Unicode 编码	239	7.3.2 创建一个新数据库	279
6.7.8 增量编码	240	7.3.3 打开一个现有数据库	279
6.7.9 Unicode 数据和网络通信	242	7.3.4 错误情况	280
6.7.10 定义定制编码	245	7.4 whichdb——识别 DBM 数据库格式	281
6.8 StringIO——提供类文件 API 的文本 缓冲区	251	7.5 sqlite3——嵌入式关系数据库	281
6.9 fnmatch——UNIX 式 glob 模式匹配	252	7.5.1 创建数据库	282
6.9.1 简单匹配	252	7.5.2 获取数据	285
6.9.2 过滤	253	7.5.3 查询元数据	286
6.9.3 转换模式	254	7.5.4 行对象	287
6.10 dircache——缓存目录列表	254	7.5.5 查询中使用变量	288
6.10.1 列出目录内容	255	7.5.6 批量加载	290
6.10.2 标注列表	256	7.5.7 定义新列类型	291
6.11 filecmp——比较文件	257	7.5.8 确定列类型	294
6.11.1 示例数据	258	7.5.9 事务	296
6.11.2 比较文件	260	7.5.10 隔离级别	298
6.11.3 比较目录	261	7.5.11 内存中数据库	302
6.11.4 程序中使用差异	262	7.5.12 导出数据库内容	302
		7.5.13 SQL 中使用 Python 函数	304

7.5.14 定制聚集	306	8.2.2 读压缩数据	349
7.5.15 定制排序	307	8.2.3 处理流	350
7.5.16 线程和连接共享	308	8.3 bz2——bzip2 压缩	352
7.5.17 限制对数据的访问	309	8.3.1 内存中一次性操作	352
7.6 xml.etree.ElementTree——XML 操纵		8.3.2 增量压缩和解压缩	354
API	311	8.3.3 混合内容流	354
7.6.1 解析 XML 文档	312	8.3.4 写压缩文件	355
7.6.2 遍历解析树	313	8.3.5 读压缩文件	357
7.6.3 查找文档中的节点	314	8.3.6 压缩网络数据	358
7.6.4 解析节点属性	315	8.4 tarfile——Tar 归档访问	362
7.6.5 解析时监视事件	317	8.4.1 测试 Tar 文件	362
7.6.6 创建一个定制树构造器	319	8.4.2 从归档文件读取元数据	362
7.6.7 解析串	321	8.4.3 从归档抽取文件	364
7.6.8 用元素节点构造文档	322	8.4.4 创建新归档	365
7.6.9 美观打印 XML	323	8.4.5 使用候选归档成员名	366
7.6.10 设置元素属性	325	8.4.6 从非文件源写数据	366
7.6.11 由节点列表构造树	327	8.4.7 追加到归档	367
7.6.12 将 XML 串行化至一个流	329	8.4.8 处理压缩归档	368
7.7 csv——逗号分隔值文件	331	8.5 zipfile——ZIP 归档访问	369
7.7.1 读文件	332	8.5.1 测试 ZIP 文件	369
7.7.2 写文件	332	8.5.2 从归档读取元数据	369
7.7.3 方言	334	8.5.3 从归档抽取归档文件	371
7.7.4 使用字段名	338	8.5.4 创建新归档	371
第 8 章 数据压缩与归档	340	8.5.5 使用候选归档成员名	373
8.1 zlib——GNU zlib 压缩	340	8.5.6 从非文件源写数据	373
8.1.1 处理内存中数据	340	8.5.7 利用 ZipInfo 实例写	374
8.1.2 增量压缩与解压缩	341	8.5.8 追加到文件	375
8.1.3 混合内容流	342	8.5.9 Python ZIP 归档	376
8.1.4 校验和	343	8.5.10 限制	377
8.1.5 压缩网络数据	343	第 9 章 加密	378
8.2 gzip——读写 GNU Zip 文件	347	9.1 hashlib——密码散列	378
8.2.1 写压缩文件	348	9.1.1 示例数据	378

9.1.2 MD5 示例	379	10.3.9 同步线程	421
9.1.3 SHA1 示例	379	10.3.10 限制资源的并发访问	422
9.1.4 按名创建散列	379	10.3.11 线程特定数据	423
9.1.5 增量更新	380	10.4 multiprocessing——像线程一样	
9.2 hmac——密码消息签名与验证	381	管理进程	425
9.2.1 消息签名	381	10.4.1 multiprocessing 基础	426
9.2.2 SHA 与 MD5	382	10.4.2 可导入的目标函数	427
9.2.3 二进制摘要	383	10.4.3 确定当前进程	428
9.2.4 消息签名的应用	383	10.4.4 守护进程	428
第 10 章 进程与线程	387	10.4.5 等待进程	430
10.1 subprocess——创建附加进程	387	10.4.6 终止进程	431
10.1.1 运行外部命令	388	10.4.7 进程退出状态	432
10.1.2 直接处理管道	391	10.4.8 日志	434
10.1.3 连接管道段	393	10.4.9 派生进程	435
10.1.4 与其他命令交互	394	10.4.10 向进程传递消息	435
10.1.5 进程间传递信号	396	10.4.11 进程间信号传输	438
10.2 signal——异步系统事件	400	10.4.12 控制资源访问	439
10.2.1 接收信号	400	10.4.13 同步操作	440
10.2.2 获取注册的处理程序	401	10.4.14 控制资源的并发访问	441
10.2.3 发送信号	402	10.4.15 管理共享状态	443
10.2.4 闹铃	403	10.4.16 共享命名空间	444
10.2.5 忽略信号	403	10.4.17 进程池	445
10.2.6 信号和线程	404	10.4.18 实现 MapReduce	447
10.3 threading——管理并发操作	406	第 11 章 网络通信	452
10.3.1 Thread 对象	406	11.1 socket——网络通信	452
10.3.2 确定当前线程	407	11.1.1 寻址、协议簇和套接字类型	452
10.3.3 守护与非守护线程	409	11.1.2 TCP/IP 客户和服务	460
10.3.4 列举所有线程	411	11.1.3 用户数据报客户和服务	467
10.3.5 派生线程	412	11.1.4 UNIX 域套接字	469
10.3.6 定时器线程	414	11.1.5 组播	473
10.3.7 线程间传送信号	415	11.1.6 发送二进制数据	476
10.3.8 控制资源访问	416	11.1.7 非阻塞通信和超时	478

11.2 select——高效等待 I/O	479	12.2.3 线程与进程	522
11.2.1 使用 select()	479	12.2.4 处理错误	523
11.2.2 有超时的非阻塞 I/O	484	12.2.5 设置首部	524
11.2.3 使用 poll()	486	12.3 urllib——网络资源访问	525
11.2.4 平台特定选项	490	12.3.1 利用缓存实现简单获取	526
11.3 SocketServer——创建网络服务器	491	12.3.2 参数编码	527
11.3.1 服务器类型	491	12.3.3 路径与 URL	529
11.3.2 服务器对象	491	12.4 urllib2——网络资源访问	530
11.3.3 实现服务器	491	12.4.1 HTTP GET	530
11.3.4 请求处理器	492	12.4.2 参数编码	532
11.3.5 回应示例	492	12.4.3 HTTP POST	533
11.3.6 线程和进程	497	12.4.4 增加发出首部	534
11.4 asyncore——异步 I/O	499	12.4.5 从请求提交表单数据	535
11.4.1 服务器	500	12.4.6 上传文件	536
11.4.2 客户	501	12.4.7 创建定制协议处理器	539
11.4.3 事件循环	503	12.5 Base64——用 ASCII 编码二进制数据	541
11.4.4 处理其他事件循环	505	12.5.1 Base64 编码	541
11.4.5 处理文件	507	12.5.2 Base64 解码	542
11.5 asynchat——异步协议处理器	508	12.5.3 URL 安全的变种	543
11.5.1 消息终止符	508	12.5.4 其他编码	543
11.5.2 服务器和处理器	508	12.6 robotparser——网络蜘蛛访问控制	544
11.5.3 客户	511	12.6.1 robots.txt	545
11.5.4 集成	512	12.6.2 测试访问权限	545
第 12 章 Internet	514	12.6.3 长久蜘蛛	546
12.1 urlparse——分解 URL	514	12.7 Cookie——HTTP Cookie	547
12.1.1 解析	515	12.7.1 创建和设置 Cookie	547
12.1.2 反解析	517	12.7.2 Morsel	548
12.1.3 连接	518	12.7.3 编码值	550
12.2 BaseHTTPServer——实现 Web 服务器的基类	519	12.7.4 接收和解析 Cookie 首部	550
12.2.1 HTTP GET	519	12.7.5 候选输出格式	551
12.2.2 HTTP POST	521	12.7.6 废弃的类	552

12.8 uuid——全局惟一标识符.....	552	第 13 章 Email.....	587
12.8.1 UUID 1——IEEE 802 MAC 地址.....	552	13.1 smtp lib——简单邮件传输协议客户.....	587
12.8.2 UUID 3 和 5——基于名字的值.....	554	13.1.1 发送 Email 消息.....	587
12.8.3 UUID 4——随机值.....	556	13.1.2 认证和加密.....	589
12.8.4 处理 UUID 对象.....	556	13.1.3 验证 Email 地址.....	592
12.9 json——JavaScript 对象记法.....	557	13.2 smtpd——示例邮件服务器.....	593
12.9.1 编码和解码简单数据类型.....	557	13.2.1 邮件服务器基类.....	593
12.9.2 优质输出和紧凑输出.....	558	13.2.2 调试服务器.....	595
12.9.3 编码字典.....	560	13.2.3 代理服务器.....	596
12.9.4 处理定制类型.....	561	13.3 imap lib——IMAP4 客户库.....	596
12.9.5 编码器和解码器类.....	563	13.3.1 变种.....	597
12.9.6 处理流和文件.....	565	13.3.2 连接到服务器.....	597
12.9.7 混合数据流.....	566	13.3.3 示例配置.....	598
12.10 xmlrpc lib——XML-RPC 的客户 端库.....	567	13.3.4 列出邮箱.....	599
12.10.1 连接服务器.....	568	13.3.5 邮箱状态.....	601
12.10.2 数据类型.....	570	13.3.6 选择邮箱.....	602
12.10.3 传递对象.....	573	13.3.7 搜索消息.....	603
12.10.4 二进制数据.....	573	13.3.8 搜索规则.....	604
12.10.5 异常处理.....	575	13.3.9 获取消息.....	605
12.10.6 将调用结合在一个消息中.....	575	13.3.10 完整消息.....	608
12.11 SimpleXMLRPCServer——一个 XML-RPC 服务器.....	577	13.3.11 上传消息.....	609
12.11.1 一个简单的服务器.....	577	13.3.12 移动和复制消息.....	611
12.11.2 备用 API 名.....	578	13.3.13 删除消息.....	612
12.11.3 加点的 API 名.....	579	13.4 mailbox——管理邮件归档.....	614
12.11.4 任意 API 名.....	580	13.4.1 mbox.....	614
12.11.5 公布对象的方法.....	581	13.4.2 Maildir.....	616
12.11.6 分派调用.....	583	13.4.3 其他格式.....	622
12.11.7 自省 API.....	584	第 14 章 应用构建模块.....	623
		14.1 getopt——命令行选项解析.....	624
		14.1.1 函数参数.....	624

14.1.2 短格式选项.....	624	14.6.1 处理命令.....	680
14.1.3 长格式选项.....	625	14.6.2 命令参数.....	681
14.1.4 一个完整的例子.....	625	14.6.3 现场帮助.....	682
14.1.5 缩写长格式选项.....	627	14.6.4 自动完成.....	683
14.1.6 GNU 选项解析.....	627	14.6.5 覆盖基类方法.....	684
14.1.7 结束参数处理.....	629	14.6.6 通过属性配置 Cmd.....	686
14.2 optparse——命令行选项解析器.....	629	14.6.7 运行 shell 命令.....	687
14.2.1 创建 OptionParser.....	629	14.6.8 候选输入.....	688
14.2.2 短格式和长格式选项.....	630	14.6.9 sys.argv 的命令.....	689
14.2.3 用 getopt 比较.....	631	14.7 shlex——解析 shell 语法.....	690
14.2.4 选项值.....	632	14.7.1 加引号的字符串.....	691
14.2.5 选项动作.....	635	14.7.2 嵌入注释.....	692
14.2.6 帮助消息.....	639	14.7.3 分解.....	693
14.3 argparse——命令行选项和参数 解析.....	644	14.7.4 包含其他 Token 源.....	693
14.3.1 与 optparse 比较.....	644	14.7.5 控制解析器.....	694
14.3.2 建立解析器.....	644	14.7.6 错误处理.....	696
14.3.3 定义参数.....	644	14.7.7 POSIX 与非 POSIX 解析.....	697
14.3.4 解析命令行.....	645	14.8 ConfigParser——处理配置文件.....	698
14.3.5 简单示例.....	645	14.8.1 配置文件格式.....	699
14.3.6 自动生成的选项.....	652	14.8.2 读取配置文件.....	699
14.3.7 解析器组织.....	653	14.8.3 访问配置设置.....	701
14.3.8 高级参数处理.....	659	14.8.4 修改设置.....	705
14.4 readline——GNU Readline 库.....	666	14.8.5 保存配置文件.....	706
14.4.1 配置.....	667	14.8.6 选项搜索路径.....	707
14.4.2 完成文本.....	668	14.8.7 用接合合并值.....	709
14.4.3 访问完成缓冲区.....	670	14.9 日志——报告状态、错误和信息 消息.....	712
14.4.4 输入历史.....	674	14.9.1 应用与库中的日志记录.....	712
14.4.5 hook.....	676	14.9.2 记入文件.....	712
14.5 getpass——安全密码提示.....	677	14.9.3 旋转日志文件.....	713
14.5.1 示例.....	677	14.9.4 详细级别.....	714
14.5.2 无终端使用 getpass.....	678	14.9.5 命名日志记录器实例.....	715
14.6 cmd——面向行的命令处理器.....	679	14.10 fileinput——命令行过滤器框架.....	716

XVIII

14.10.1	M3U 文件转换为 RSS	716	16.1.3	交互式帮助	746
14.10.2	进度元数据	718	16.2	doctest——通过文档完成测试	747
14.10.3	原地过滤	719	16.2.1	开始	747
14.11	atexit——程序关闭回调	721	16.2.2	处理不可预测的输出	748
14.11.1	示例	721	16.2.3	Traceback	752
14.11.2	什么情况下不调用 atexit 函数	722	16.2.4	避开空白符	753
14.11.3	处理异常	724	16.2.5	测试位置	758
14.12	sched——定时事件调度器	725	16.2.6	外部文档	761
14.12.1	有延迟地运行事件	725	16.2.7	运行测试	763
14.12.2	重叠事件	726	16.2.8	测试上下文	766
14.12.3	事件优先级	727	16.3	unittest——自动测试框架	769
14.12.4	取消事件	727	16.3.1	基本测试结构	769
第 15 章	国际化和本地化	729	16.3.2	运行测试	770
15.1	gettext——消息编目	729	16.3.3	测试结果	770
15.1.1	转换 workflow 概述	729	16.3.4	断言真值	772
15.1.2	由源代码创建消息编目	730	16.3.5	测试相等性	773
15.1.3	运行时查找消息编目	732	16.3.6	近似相等	774
15.1.4	复数值	733	16.3.7	测试异常	775
15.1.5	应用与模块本地化	735	16.3.8	测试固件	775
15.1.6	切换转换	736	16.3.9	测试套件	776
15.2	locale——文化本地化 API	736	16.4	traceback——异常和栈轨迹	777
15.2.1	探查当前本地化环境	737	16.4.1	支持函数	777
15.2.2	货币	742	16.4.2	处理异常	777
15.2.3	格式化数字	742	16.4.3	处理栈	780
15.2.4	解析数字	743	16.5	cgitb——详细的 traceback 报告	783
15.2.5	日期和时间	744	16.5.1	标准 traceback 转储	783
第 16 章	开发工具	745	16.5.2	启用详细 traceback	783
16.1	pydoc——模块的联机帮助	746	16.5.3	traceback 中的局部变量	785
16.1.1	纯文本帮助	746	16.5.4	异常属性	787
16.1.2	HTML 帮助	746	16.5.5	HTML 输出	788
			16.5.6	记录 traceback	789
			16.6	pdb——交互式调试工具	791

16.6.1 启动调试工具.....	791	第 17 章 运行时特性.....	847
16.6.2 控制调试工具.....	794	17.1 site——全站点配置.....	847
16.6.3 断点.....	803	17.1.1 导入路径.....	847
16.6.4 改变执行流.....	813	17.1.2 用户目录.....	849
16.6.5 用别名定制调试工具.....	819	17.1.3 路径配置文件.....	850
16.6.6 保存配置设置.....	821	17.1.4 定制站点配置.....	852
16.7 trace——执行程序流.....	822	17.1.5 定制用户配置.....	853
16.7.1 示例程序.....	822	17.1.6 禁用 site 模块.....	854
16.7.2 跟踪执行.....	822	17.2 sys——系统特定的配置.....	854
16.7.3 代码覆盖.....	823	17.2.1 解释器设置.....	855
16.7.4 调用关系.....	825	17.2.2 运行时环境.....	860
16.7.5 编程接口.....	826	17.2.3 内存管理和限制.....	862
16.7.6 保存结果数据.....	828	17.2.4 异常处理.....	867
16.7.7 选项.....	829	17.2.5 底层线程支持.....	869
16.8 profile 和 pstats——性能分析.....	830	17.2.6 模块和导入.....	875
16.8.1 运行性能分析工具.....	830	17.2.7 跟踪程序运行情况.....	892
16.8.2 在上下文中运行.....	832	17.3 os——可移植访问操作系统特定	
16.8.3 pstats: 保存和处理统计信息.....	833	特性.....	898
16.8.4 限制报告内容.....	835	17.3.1 进程所有者.....	898
16.8.5 调用图.....	836	17.3.2 进程环境.....	900
16.9 timeit——测量小段 Python 代码的		17.3.3 进程工作目录.....	901
执行时间.....	837	17.3.4 管道.....	901
16.9.1 模块内容.....	837	17.3.5 文件描述符.....	905
16.9.2 基本示例.....	837	17.3.6 文件系统权限.....	905
16.9.3 值存储在字典中.....	838	17.3.7 目录.....	906
16.9.4 从命令行执行.....	840	17.3.8 符号链接.....	907
16.10 compileall——字节编译源文件.....	841	17.3.9 遍历目录树.....	907
16.10.1 编译一个目录.....	842	17.3.10 运行外部命令.....	909
16.10.2 编译 sys.path.....	842	17.3.11 用 os.fork() 创建进程.....	910
16.10.3 从命令行执行.....	843	17.3.12 等待子进程.....	911
16.11 pycldr——类浏览器.....	843	17.3.13 Spawn.....	913
16.11.1 扫描类.....	845	17.3.14 文件系统权限.....	913
16.11.2 扫描函数.....	846		

17.4 platform——系统版本信息	914	18.2.5 abc 中的具体方法	956
17.4.1 解释器	915	18.2.6 抽象属性	957
17.4.2 平台	916	18.3 dis——Python 字节码反汇编	
17.4.3 操作系统和硬件信息	916	工具	960
17.4.4 可执行程序体系结构	918	18.3.1 基本反汇编	961
17.5 resource——系统资源管理	918	18.3.2 反汇编函数	961
17.5.1 当前使用情况	919	18.3.3 类	963
17.5.2 资源限制	919	18.3.4 使用反汇编进行调试	963
17.6 gc——垃圾回收器	922	18.3.5 循环的性能分析	965
17.6.1 跟踪引用	922	18.3.6 编译器优化	970
17.6.2 强制垃圾回收	925	18.4 inspect——检查现场对象	972
17.6.3 查找无法收集的对象引用	928	18.4.1 示例模块	972
17.6.4 回收阈限和代	931	18.4.2 模块信息	973
17.6.5 调试	933	18.4.3 检查模块	974
17.7 sysconfig——解释器编译时配置	940	18.4.4 检查类	975
17.7.1 配置变量	940	18.4.5 文档串	976
17.7.2 安装路径	942	18.4.6 获取源代码	977
17.7.3 Python 版本和平台	945	18.4.7 方法和函数参数	979
第 18 章 语言工具	947	18.4.8 类层次结构	980
18.1 warnings——非致命警告	947	18.4.9 方法解析顺序	981
18.1.1 分类和过滤	948	18.4.10 栈与帧	982
18.1.2 生成警告	948	18.5 exceptions——内置异常类	984
18.1.3 用模式过滤	949	18.5.1 基类	985
18.1.4 重复的警告	951	18.5.2 产生的异常	985
18.1.5 候选消息传送函数	951	18.5.3 警告类型	998
18.1.6 格式化	952	第 19 章 模块与包	999
18.1.7 警告中的栈层次	952	19.1 imp——Python 的导入机制	999
18.2 abc——抽象基类	953	19.1.1 示例包	999
18.2.1 为什么使用抽象基类	953	19.1.2 模块类型	999
18.2.2 抽象基类如何工作	954	19.1.3 查找模块	1000
18.2.3 注册一个具体类	954	19.1.4 加载模块	1001
18.2.4 通过派生实现	955	19.2 zipimport——从 ZIP 归档加载	

Python 代码.....	1003	19.3 pkgutil——包工具.....	1008
19.2.1 示例.....	1003	19.3.1 包导入路径.....	1008
19.2.2 查找模块.....	1004	19.3.2 包的开发版本.....	1010
19.2.3 访问代码.....	1004	19.3.3 用 PKG 文件管理路径.....	1011
19.2.4 源代码.....	1005	19.3.4 嵌套包.....	1013
19.2.5 包.....	1006	19.3.5 包数据.....	1014
19.2.6 数据.....	1006		



第 ① 章

文 本

对 Python 程序员来说，最显而易见的文本处理工具就是 `string` 类，不过除此以外，标准库中还提供了大量其他工具，可以帮你轻松地完成高级文本处理。

用 Python 2.0 之前版本编写的老代码使用的是 `string` 模块的函数，而不是 `string` 对象的方法。对应这个模块中的每一个函数都有一个等价的方法，新代码已经不再使用那些函数。

使用 Python 2.4 或以后版本的程序可能会使用 `string.Template` 作为一个简便方法，除了具备 `string` 或 `unicode` 类的特性，还可以对字符串实现参数化。与很多 Web 框架定义的模板或 Python Package Index 提供的扩展模块相比，尽管 `string.Template` 没有那么丰富的特性，但作为用户可修改的模板，即需要在静态文本中插入动态值，它确实很好地做到了二者兼顾。

`textwrap` 模块包括一些工具，可以对从段落中抽取的文本进行格式化，如限制输出的宽度、增加缩进，以及插入换行符从而能一致地自动换行。

除了 `string` 对象支持的内置相等性和排序比较之外，标准库还包括两个与比较文本值有关的模块。`re` 提供了一个完整的正则表达式库，出于速度原因这个库使用 C 实现。正则表达式非常适合在较大的数据集中查找子串，能够根据比固定字符串更为复杂的模式比较字符串，还可以完成一定程度的解析。

另一方面，`diff` 模块会根据添加、删除或修改的部分来计算不同文本序列之间的具体差别。`diff` 中比较函数的输出可以用来为用户提供更详细的反馈，指出两个输入中出现变化的地方，一个文档随时间有哪些改变，等等。

1.1 `string`——文本常量和模板

作用：包含处理文本的常量和类。

Python 版本：1.4 及以后版本

`string` 模块可以追溯到最早的 Python 版本。到了 2.0 版本，原先仅在这个模块中实现的很多函数则被移植为 `str` 和 `unicode` 对象的方法。仍然在遗留代码中使用这些函数，不过如今这些函数已经废弃，将在 Python 3.0 中完全去除。`string` 模块保留了很多有用的常量和类，用来处理 `string` 和 `unicode` 对象，这里就重点讨论这些常量和类。

1.1.1 函数

还有两个函数未从 `string` 模块移出：`capwords()` 和 `maketrans()`。`capwords()` 的作用是将一

个字符串中所有单词的首字母大写。

```
import string

s = 'The quick brown fox jumped over the lazy dog.'

print s
print string.capwords(s)
```

其结果等同于先调用 `split()`，这会将结果列表中各个单词的首字母大写，然后调用 `join()` 合并结果。

```
$ python string_capwords.py

The quick brown fox jumped over the lazy dog.
The Quick Brown Fox Jumped Over The Lazy Dog.
```

`maketrans()` 函数将创建转换表，可以用来结合 `translate()` 方法将一组字符修改为另一组字符，这种做法比反复调用 `replace()` 更为高效。

```
import string

leet = string.maketrans('abegiloprstz', '463611092572')
s = 'The quick brown fox jumped over the lazy dog.'

print s
print s.translate(leet)
```

在这个例子中，一些字母被替换为相应的“火星文”数字^①。

```
$ python string_maketrans.py

The quick brown fox jumped over the lazy dog.
Th3 qulck 620wn f0x jum93d 0v32 7h3 142y d06.
```

1.1.2 模板

字符串模板已经作为 PEP 292 的一部分增加到 Python 2.4 中，并得到扩展，成为替代内置拼接（interpolation）^②语法的一种候选方法。使用 `string.Template` 拼接时，可以在变量名前面加上前缀 `$`（如 `$var`）来标识变量，或者如果需要与两侧的文本相区分，还可以用大括号将变量括起（如 `${var}`）。

① Leet (L337、3L337、31337、leetspeak、eleet、Leetors、L3370rz 或 l337)，火星文，又称黑客语，是指一种发源于欧美地区的 BBS、线上游戏和黑客社群所使用的文字书写方式。通常是把拉丁字母转变成数字或是特殊符号，例如 E 写成 3、A 写成 @ 等。或将单字写成同音的字母或数字，如 to 写成 2、for 写成 4 等。——译者注

② interpolation 也称为“连接”、“插补”或“替换”，是指在文本部分插入变量或表达式的值来拼接字符串。——译者注

下面的例子对一个简单的模板和一个使用 % 操作符的类似字符串拼接进行了比较。

```
import string

values = { 'var': 'foo' }

t = string.Template("""
Variable      : $var
Escape       : $$
Variable in text: ${var}iable
""")

print 'TEMPLATE:', t.substitute(values)

s = """
Variable      : %(var)s
Escape       : %%
Variable in text: %(var)siable
"""

print 'INTERPOLATION:', s % values
```

在这两种情况下，触发器字符（\$ 或 %）都要写两次来完成转义。

```
$ python string_template.py
```

```
TEMPLATE:
Variable      : foo
Escape       : $
Variable in text: fooiable

INTERPOLATION:
Variable      : foo
Escape       : %
Variable in text: fooiable
```

模板与标准字符串拼接有一个重要区别，即模板不考虑参数类型。值会转换为字符串，再将字符串插入到结果中。这里没有提供格式化选项。例如，没有办法控制使用几位有效数字来表示一个浮点数值。

不过，这也有一个好处：通过使用 `safe_substitute()` 方法，可以避免未能提供模板所需全部参数值时可能产生的异常。

```
import string

values = { 'var': 'foo' }

t = string.Template("$var is here but $missing is not provided")

try:
    print 'substitute()      :', t.substitute(values)
```

```

except KeyError, err:
    print 'ERROR:', str(err)

print 'safe_substitute():', t.safe_substitute(values)

```

由于 values 字典中没有对应 missing 的值，因此 substitute() 会产生一个 KeyError。不过，safe_substitute() 不会抛出这个错误，它将捕获这个异常，并在文本中保留变量表达式。

```

$ python string_template_missing.py
substitute()      : ERROR: 'missing'
safe_substitute(): foo is here but $missing is not provided

```

1.1.3 高级模板

可以修改 string.Template 的默认语法，为此要调整它在模板体中查找变量名所使用的正则表达式模式。一种简单的做法是修改 delimiter 和 idpattern 类属性。

```

import string

template_text = '''
    Delimiter : %%
    Replaced  : %with_underscore
    Ignored   : %notunderscored
'''

d = { 'with_underscore': 'replaced',
      'notunderscored': 'not replaced',
      }

class MyTemplate(string.Template):
    delimiter = '%'
    idpattern = '[a-z]_[a-z]+'

t = MyTemplate(template_text)
print 'Modified ID pattern:'
print t.safe_substitute(d)

```

在这个例子中，替换规则已经改变，定界符是 % 而不是 \$，另外变量名必须包含一条下划线。模式 %notunderscored 未得到替换，因为其中不包含下划线字符。

```
$ python string_template_advanced.py
```

```
Modified ID pattern:
```

```

Delimiter : %
Replaced  : replaced
Ignored   : %notunderscored

```

要完成更复杂的修改，可以覆盖 `pattern` 属性，定义一个全新的正则表达式。所提供的模式必须包含 4 个命名组，分别对应转义定界符、命名变量、用大括号括住的变量名，以及不合法的定界符模式。

```
import string
```

```
t = string.Template('$var')
print t.pattern.pattern
```

`t.pattern` 是一个已编译的正则表达式，不过可以通过其 `pattern` 属性得到原来的字符串表示。

```
\$(?:
    (?P<escaped>\$) |                # two delimiters
    (?P<named>[_a-z][_a-z0-9]*)      | # identifier
    {(?P<braced>[_a-z][_a-z0-9]*)} | # braced identifier
    (?P<invalid>)                   # ill-formed delimiter exprs
)
```

下面的例子定义了一个新模式来创建一个新的模板类型：使用 `{{var}}` 作为变量语法。

```
import re
```

```
import string
```

```
class MyTemplate(string.Template):
    delimiter = '{{'
    pattern = re.compile(
        r'\{\{(?:
            (?P<escaped>\{\{})|
            (?P<named>[_a-z][_a-z0-9]*)\}\}|
            (?P<braced>[_a-z][_a-z0-9]*)\}\}\}|
            (?P<invalid>)
        )
    )
```

```
t = MyTemplate('{{
{{{
{{var}}
}}')
print 'MATCHES:', t.pattern.findall(t.template)
print 'SUBSTITUTED:', t.safe_substitute(var='replacement')
```

`named` 和 `braced` 模式必须单独提供，尽管它们实际上是一样的。运行这个示例程序会生成以下结果：

```
$ python string_template_newsyntax.py
```

```
MATCHES: [('{{', '{{', '{{', '{{'), ('', 'var', '', '')]
SUBSTITUTED:
{{
replacement
```

参见：

string (<http://docs.python.org/lib/module-string.html>) 这个模块的标准库文档。

String Methods (<http://docs.python.org/lib/string-methods.html#string-methods>) str 对象的方法，取代已经废弃的 string 函数。

PEP 292 (www.python.org/dev/peps/pep-0292) 一种更简单的字符串替换语法的提案。

L33t (<http://en.wikipedia.org/wiki/Leet>) “火星文”字母表。

1.2 textwrap——格式化文本段落

作用：通过调整换行符在段落中出现的位置来格式化文本。

Python 版本：2.5 及以后版本

需要美观打印 (pretty-printing) 时，可以用 textwrap 模块来格式化要输出的文本。这个模块允许通过编程提供类似段落自动换行或填充特性等功能（很多文本编辑器和字处理器都提供有这些功能）。

1.2.1 示例数据

本节中的例子使用了模块 textwrap_example.py，其中包含一个字符串 sample_text。

```
sample_text = '''
    The textwrap module can be used to format text for output in
    situations where pretty-printing is desired. It offers
    programmatic functionality similar to the paragraph wrapping
    or filling features found in many text editors.
'''
```

1.2.2 填充段落

fill() 函数取文本作为输入，生成格式化的文本作为输出。

```
import textwrap
from textwrap_example import sample_text

print 'No dedent:\n'
print textwrap.fill(sample_text, width=50)
```

这个结果还有些差强人意。现在文本是左对齐的，不过只有第一行保留了缩进，其余各行前面的空格则嵌入到段落中。

```
$ python textwrap_fill.py
```

```
No dedent:
```

```
    The textwrap module can be used to format
text for output in      situations where pretty-
```

```
printing is desired. It offers      programmatic
functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

1.2.3 去除现有缩进

在前面的例子中，输出里混合嵌入了制表符和额外的空格，所以格式不太美观。从示例文本删除所有行中都有的空白符前缀可以生成更好的结果，从而能直接使用 Python 代码中的 `docstring` 或嵌入的多行字符串，同时自行去除代码的格式化。示例字符串人为地引入了一级缩进，以便展示这个特性。

```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text)
print 'Dedented:'
print dedented_text
```

结果变得漂亮一些了：

```
$ python textwrap_dedent.py
```

Dedented:

```
The textwrap module can be used to format text for output in
situations where pretty-printing is desired. It offers
programmatic functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

由于“`dedent`”（去除缩进）与“`indent`”（缩进）正好相反，因此这里的结果是得到一个文本块，而且删除了各行最前面都有的空白符。如果某一行比其他行缩进更多，则会有一些空白符未删除。

以下输入：

```
_Line one.
__Line two.
__Line three.
```

会变成：

```
Line one.
__Line two.
Line three.
```

1.2.4 结合 `dedent` 和 `fill`

接下来，可以把去除缩进的文本传入 `fill()`，并提供一组不同的 `width` 值。



```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text).strip()
for width in [ 45, 70 ]:
    print '%d Columns:\n' % width
    print textwrap.fill(dedented_text, width=width)
    print
```

这会生成指定宽度的输出。

```
$ python textwrap_fill_width.py
```

45 Columns:

The textwrap module can be used to format text for output in situations where pretty-printing is desired. It offers programmatic functionality similar to the paragraph wrapping or filling features found in many text editors.

70 Columns:

The textwrap module can be used to format text for output in situations where pretty-printing is desired. It offers programmatic functionality similar to the paragraph wrapping or filling features found in many text editors.

1.2.5 悬挂缩进

不仅输出的宽度可以设置，还可以单独控制第一行的缩进，以区别后面各行。

```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text).strip()
print textwrap.fill(dedented_text,
                    initial_indent='',
                    subsequent_indent=' ' * 4,
                    width=50,
                    )
```

这样一来会生成一种悬挂缩进，即第一行的缩进小于其他行的缩进。

```
$ python textwrap_hanging_indent.py
```

The textwrap module can be used to format text for




```
output in situations where pretty-printing is
desired. It offers programmatic functionality
similar to the paragraph wrapping or filling
features found in many text editors.
```

缩进值还可以包含非空白符。例如，悬挂缩进可以加前缀 * 来生成项目符号。

参见：

`textwrap` (<http://docs.python.org/lib/module-textwrap.html>) 这个模块的标准库文档。

1.3 re——正则表达式

作用：使用形式化模式搜索和修改文本。

Python 版本：1.5 及以后版本

正则表达式 (regular expression) 是用一种形式化语法描述的文本匹配模式。模式被解释为一组指令，然后会执行这组指令，以一个字符串作为输入，生成一个匹配的子集或原字符串的修改版本。“正则表达式”一词在讨论中通常会简写为“regex”或“regexp”。表达式可以包括字面量文本匹配、重复、模式组合、分支以及其他复杂的规则。对于很多解析问题，用正则表达式解决会比创建特殊用途的词法分析器和语法分析器更为容易。

正则表达式通常在涉及大量文本处理的应用中使用。例如，在开发人员使用的文本编辑程序中（包括 vi、emacs 和其他现代 IDE）常用正则表达式作为搜索模式。另外，正则表达式还是 UNIX 命令行工具的一个不可缺少的部分，如 sed、grep 和 awk。很多编程语言都在语言语法中包括对正则表达式的支持，如 Perl、Ruby、Awk 和 Tcl。另外一些语言（如 C、C++ 和 Python）则通过扩展库来支持正则表达式。

有很多开源的正则表达式实现，这些实现都有一种共同的核心语法，不过对其高级特性有不同的扩展或修改。Python 的 re 模块中使用的语法以 Perl 所用正则表达式语法为基础，并提供了一些特定于 Python 的改进。

注意：尽管“正则表达式”的正式定义仅限于描述正则语言的表达式，但 re 支持的一些扩展已不仅仅描述正则语言。这里“正则表达式”一词有更通用的含义，表示可以由 Python 的 re 模块计算的所有表达式。

1.3.1 查找文本中的模式

re 最常见的用法就是搜索文本中的模式。`search()` 函数取模式和要扫描的文本作为输入，如果找到这个模式则返回一个 Match 对象。如果未找到模式，`search()` 将返回 None。

每个 Match 对象包含有关匹配性质的信息，包括原输入字符串、使用的正则表达式，以及模式在原字符串中出现的位置。

```
import re
```

```
pattern = 'this'
text = 'Does this text match the pattern?'

match = re.search(pattern, text)

s = match.start()
e = match.end()

print 'Found "%s" in "%s" from %d to %d ("%s")' % \
      (match.re.pattern, match.string, s, e, text[s:e])
```

start() 和 end() 方法可以给出字符串中的相应索引，指示与模式匹配的文本在字符串中出现的位置。

```
$ python re_simple_match.py

Found "this"
in "Does this text match the pattern?"
from 5 to 9 ("this")
```

1.3.2 编译表达式

re 包含一些模块级函数，用于处理作为文本字符串的正则表达式，不过对于程序频繁使用的表达式，编译这些表达式会更为高效。compile() 函数会把一个表达式字符串转换为一个 `RegexObject`。

```
import re

# Precompile the patterns
regexes = [ re.compile(p)
             for p in [ 'this', 'that' ]
            ]

text = 'Does this text match the pattern?'

print 'Text: %r\n' % text

for regex in regexes:
    print 'Seeking "%s" ->' % regex.pattern,

    if regex.search(text):
        print 'match!'
    else:
        print 'no match'
```

模块级函数会维护已编译表达式的一个缓存。不过，这个缓存的大小是有限的，直接使用已编译表达式可以避免缓存查找开销。使用已编译表达式的另一个好处是，通过在加载模块时预编译所有表达式，可以把编译工作转到应用开始时，而不是当程序响应一个用户动作时才进

行编译。

```
$ python re_simple_compiled.py

Text: 'Does this text match the pattern?'

Seeking "this" -> match!
Seeking "that" -> no match
```

1.3.3 多重匹配

到目前为止，示例模式都使用 `search()` 来查找字面量文本字符串的单个实例。`findall()` 函数会返回输入中与模式匹配而不重叠的所有子串。

```
import re

text = 'abbbaabbbbbaaaaa'

pattern = 'ab'

for match in re.findall(pattern, text):
    print 'Found "%s"' % match
```

这个输入字符串中有两个 `ab` 实例。

```
$ python re_findall.py
```

```
Found "ab"
Found "ab"
```

`finditer()` 会返回一个迭代器，它将生成 `Match` 实例，而不像 `findall()` 返回字符串。

```
import re

text = 'abbbaabbbbbaaaaa'

pattern = 'ab'

for match in re.finditer(pattern, text):
    s = match.start()
    e = match.end()
    print 'Found "%s" at %d:%d' % (text[s:e], s, e)
```

这个例子找到了 `ab` 的两次出现，`Match` 实例显示出它们在原输入字符串中的位置。

```
$ python re_finditer.py
```

```
Found "ab" at 0:2
Found "ab" at 5:7
```

1.3.4 模式语法

正则表达式支持更强大的模式，而不只是简单的字面量文本字符串。模式可以重复，可以锚定到输入中不同的逻辑位置，还可以采用紧凑形式表示而不需要在模式中提供每一个字面量字符。使用所有这些特性时，需要结合字面量文本值和元字符 (metacharacter)，元字符是 re 实现的正则表达式模式语法的一部分。

```
import re

def test_patterns(text, patterns=[]):
    """Given source text and a list of patterns, look for
    matches for each pattern within the text and print
    them to stdout.
    """
    # Look for each pattern in the text and print the results
    for pattern, desc in patterns:
        print 'Pattern %r (%s)\n' % (pattern, desc)
        print ' %r' % text
        for match in re.finditer(pattern, text):
            s = match.start()
            e = match.end()
            substr = text[s:e]
            n_backslashes = text[:s].count('\\')
            prefix = '.' * (s + n_backslashes)
            print ' %s%sr' % (prefix, substr)
        print
    return

if __name__ == '__main__':
    test_patterns('abbbaabbbbbaaaaa',
                  [('ab', "'a' followed by 'b'"),
                   ])

```

下面的例子将使用 test_patterns() 来研究模式变化如何影响以何种方式匹配相同的输入文本。输出显示了输入文本以及输入中与模式匹配的各个部分的子串区间。

```
$ python re_test_patterns.py

Pattern 'ab' ('a' followed by 'b')

'abbbaabbbbbaaaaa'
'ab'
.....'ab'

```

重复

模式中有 5 种表达重复的方式。如果模式后面跟有元字符 *，这个模式会重复 0 次或多次。

(允许一个模式重复 0 次意味着这个模式并不需要出现就能够匹配)。如果将 * 替换为 +, 那么这个模式必须至少出现 1 次。使用 “?” 意味着模式要出现 0 或 1 次。如果希望出现特定的次数, 需要在模式后面使用 {m}, 这里 m 是模式需要重复的次数。最后, 如果允许重复次数可变但是有一个限定范围, 可以使用 {m,n}, 这里 m 是最小重复次数, n 是最大重复次数。如果省略 n (即 {m,}), 表示这个值至少要出现 m 次, 但无上限。

```
from re_test_patterns import test_patterns

test_patterns(
    'abbaabbbba',
    [ ('ab*',      'a followed by zero or more b'),
      ('ab+',      'a followed by one or more b'),
      ('ab?',      'a followed by zero or one b'),
      ('ab{3}',    'a followed by three b'),
      ('ab{2,3}',  'a followed by two to three b'),
    ])
```

ab* 和 ab? 的匹配模式要多于 ab+ 的匹配。

```
$ python re_repetition.py
```

```
Pattern 'ab*' (a followed by zero or more b)
```

```
'abbaabbbba'
'abb'
...'a'
....'abbb'
.....'a'
```

```
Pattern 'ab+' (a followed by one or more b)
```

```
'abbaabbbba'
'abb'
....'abbb'
```

```
Pattern 'ab?' (a followed by zero or one b)
```

```
'abbaabbbba'
'ab'
...'a'
....'ab'
.....'a'
```

```
Pattern 'ab{3}' (a followed by three b)
```

```
'abbaabbbba'
....'abbb'
```

```
Pattern 'ab{2,3}' (a followed by two to three b)
```



```
'abbaabbba'
'abb'
....'abbb'
```

正常情况下，处理重复指令时，re 匹配模式时会利用 (consume) 尽可能多的输入。这种所谓“贪心”的行为可能导致单个匹配减少，或者匹配中包含了多于原先预计的输入文本。在重复指令后面加上“?” 可以关闭这种贪心行为。

```
from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [ ('ab*?',      'a followed by zero or more b'),
      ('ab+?',      'a followed by one or more b'),
      ('ab??',      'a followed by zero or one b'),
      ('ab{3}?',    'a followed by three b'),
      ('ab{2,3}?',  'a followed by two to three b'),
    ])
```

对于允许 b 出现 0 次的模式，如果禁用贪心利用 (consume) 输入，意味着匹配的子串不包含任何 b 字符。

```
$ python re_repetition_non_greedy.py
```

```
Pattern 'ab*?' (a followed by zero or more b)
```

```
'abbaabbba'
'a'
...'a'
....'a'
.....'a'
```

```
Pattern 'ab+?' (a followed by one or more b)
```

```
'abbaabbba'
'ab'
....'ab'
```

```
Pattern 'ab??' (a followed by zero or one b)
```

```
'abbaabbba'
'a'
...'a'
....'a'
.....'a'
```

```
Pattern 'ab{3}?' (a followed by three b)
```

```
'abbaabbba'
```



```
....'abbb'
```

```
Pattern 'ab{2,3}?' (a followed by two to three b)
```

```
'abbaabbba'
'abb'
....'abb'
```

字符集

字符集 (character set) 是一组字符, 包含可以与模式中相应位置匹配的所有字符。例如, [ab] 可以匹配 a 或 b。

```
from re_test_patterns import test_patterns
```

```
test_patterns(
    'abbaabbba',
    [ ('[ab]',      'either a or b'),
      ('a[ab]+',    'a followed by 1 or more a or b'),
      ('a[ab]+?',   'a followed by 1 or more a or b, not greedy'),
    ])
```

表达式 (a[ab]+) 的贪心形式会利用整个字符串, 因为第一个字母是 a, 而且后面的每一个字符要么是 a 要么是 b。

```
$ python re_charset.py
```

```
Pattern '[ab]' (either a or b)
```

```
'abbaabbba'
'a'
'b'
..b'
...a'
....a'
.....b'
.....b'
.....b'
.....a'
```

```
Pattern 'a[ab]+' (a followed by 1 or more a or b)
```

```
'abbaabbba'
'abbaabbba'
```

```
Pattern 'a[ab]+?' (a followed by 1 or more a or b, not greedy)
```

```
'abbaabbba'
'ab'
...aa'
```



字符集还可以用来排除某些特定字符。尖字符 (^) 表示要查找未在随后的字符集中出现的字符。

```
from re_test_patterns import test_patterns

test_patterns(
    'This is some text -- with punctuation.',
    [ ('[^- . ]+', 'sequences without -, ., or space'),
      ])

```

这个模式将找到不包含字符 “-”、“.” 或空格的所有子串。

```
$ python re_charset_exclude.py
```

```
Pattern '[^- . ]+' (sequences without -, ., or space)
'This is some text -- with punctuation.'
'This'
.....'is'
.....'some'
.....'text'
.....'with'
.....'punctuation'

```

随着字符集变大，键入每一个应当（或不应当）匹配的字符会显得很枯燥。可以使用一种更紧凑的格式，利用字符区间（character range）来定义一个字符集，其中包括一个起点和一个终点之间所有连续的字符。

```
from re_test_patterns import test_patterns

test_patterns(
    'This is some text -- with punctuation.',
    [ ('[a-z]+', 'sequences of lowercase letters'),
      ('[A-Z]+', 'sequences of uppercase letters'),
      ('[a-zA-Z]+', 'sequences of lowercase or uppercase letters'),
      ('[A-Z][a-z]+', 'one uppercase followed by lowercase'),
      ])

```

这里的区间 a-z 包括所有小写 ASCII 字母，区间 A-Z 包括全部大写 ASCII 字母。这些区间还可以合并为一个字符集。

```
$ python re_charset_ranges.py
```

```
Pattern '[a-z]+' (sequences of lowercase letters)

'This is some text -- with punctuation.'
.'his'
.....'is'
.....'some'
.....'text'

```



```
.....'with'
.....'punctuation'
```

Pattern '[A-Z]+' (sequences of uppercase letters)

```
'This is some text -- with punctuation.'
'T'
```

Pattern '[a-zA-Z]+' (sequences of lowercase or uppercase letters)

```
'This is some text -- with punctuation.'
'This'
.....'is'
.....'some'
.....'text'
.....'with'
.....'punctuation'
```

Pattern '[A-Z][a-z]+' (one uppercase followed by lowercase)

```
'This is some text -- with punctuation.'
'This'
```

作为字符集的一种特殊情况，元字符“.”（点号）指模式应当匹配该位置的任何单字符。

```
from re_test_patterns import test_patterns
```

```
test_patterns(
    'abbaabbba',
    [ ('a.', 'a followed by any one character'),
      ('b.', 'b followed by any one character'),
      ('a.*b', 'a followed by anything, ending in b'),
      ('a.*?b', 'a followed by anything, ending in b'),
    ])
```

结合点号与重复可以得到非常长的匹配结果，除非使用非贪心形式。

```
$ python re_charset_dot.py
```

Pattern 'a.' (a followed by any one character)

```
'abbaabbba'
'ab'
...'aa'
```

Pattern 'b.' (b followed by any one character)

```
'abbaabbba'
.'bb'
.....'bb'
.....'ba'
```

Pattern 'a.*b' (a followed by anything, ending in b)

```
'abbaabbbba'
'abbaabbbb'
```

Pattern 'a.*?b' (a followed by anything, ending in b)

```
'abbaabbbba'
'ab'
...'aab'
```

转义码

还有一种更为紧凑的表示，可以对一些预定义的字符集使用转义码。re 可以识别的转义码如表 1.1 所示。

表 1.1 正则表达式转义码

转 义 码	含 义
\d	一个数字
\D	一个非数字
\s	空白符（制表符、空格、换行符等）
\S	非空白符
\w	字母数字
\W	非字母数字

注意：转义字符通过在该字符前面加一个反斜线(\)前缀来指示。遗憾的是，正常的 Python 字符串中反斜线自身也必须转义，这就会导致表达式很难阅读。通过使用“原始”(raw)字符串（在字面值前面加一个前缀 r 来创建），可以消除这个问题，并维持可读性。

```
from re_test_patterns import test_patterns

test_patterns(
    'A prime #1 example!',
    [ (r'\d+', 'sequence of digits'),
      (r'\D+', 'sequence of nondigits'),
      (r'\s+', 'sequence of whitespace'),
      (r'\S+', 'sequence of nonwhitespace'),
      (r'\w+', 'alphanumeric characters'),
      (r'\W+', 'nonalphanumeric'),
    ]
)
```

以下示例表达式结合了转义码和重复，来查找输入字符串中的类似字符序列。

```
$ python re_escape_codes.py
```

Pattern '\\d+' (sequence of digits)

```
'A prime #1 example!'
.....'1'
```

Pattern '\\D+' (sequence of nondigits)

```
'A prime #1 example!'
'A prime #'
.....' example!'
```

Pattern '\\s+' (sequence of whitespace)

```
'A prime #1 example!'
.' '
.....' '
.....' ' '
```

Pattern '\\S+' (sequence of nonwhitespace)

```
'A prime #1 example!'
'A'
..'prime'
.....'#1'
.....'example!'
```

Pattern '\\w+' (alphanumeric characters)

```
'A prime #1 example!'
'A'
..'prime'
.....'1'
.....'example'
```

Pattern '\\W+' (nonalphanumeric)

```
'A prime #1 example!'
.' '
.....' #'
.....' '
.....'!''
```

要匹配属于正则表达式语法的字符，需要对搜索模式中的字符进行转义。

```
from re_test_patterns import test_patterns
```

```
test_patterns(
    r'\d+ \D+ \s+',
```



```
[ (r'\\.\\+', 'escape code'),
]
```

这个例子中的模式对反斜线和加号字符进行了转义，因为作为元字符，这两个字符在正则表达式中都有特殊的含义。

```
$ python re_escape_escapes.py
```

```
Pattern '\\\\.\\+' (escape code)
```

```
'\\d+ \\D+ \\s+'
'\\d+'
.....'\\D+'
.....'\\s+'
```

锚定

除了描述要匹配的模式的内容外，还可以使用锚定（anchoring）指令指定输入文本中模式应当出现的相对位置。表 1.2 列出了合法的锚定码。

表 1.2 正则表达式锚定码

锚 定 码	含 义
^	字符串或行的开始
\$	字符串或行的结束
\A	字符串开始
\Z	字符串结束
\b	一个单词开头或末尾的空串
\B	不在一个单词开头或末尾的空串

```
from re_test_patterns import test_patterns
```

```
test_patterns(
    'This is some text -- with punctuation.',
    [ (r'^\w+', 'word at start of string'),
      (r'\A\w+', 'word at start of string'),
      (r'\w+\S*$', 'word near end of string, skip punctuation'),
      (r'\w+\S*\Z', 'word near end of string, skip punctuation'),
      (r'\w*t\w*', 'word containing t'),
      (r'\bt\w+', 't at start of word'),
      (r'\w+t\b', 't at end of word'),
      (r'\Bt\B', 't, not start or end of word'),
    ])
```

这个例子中，匹配字符串开头和末尾单词的模式是不同的，因为字符串末尾的单词后面有一个结束句子的标点符号。模式 `\w+$` 不能匹配，因为“.”并不是一个字母数字字符。

```
$ python re_anchoring.py
```

```
Pattern '^\\w+' (word at start of string)
```

```
'This is some text -- with punctuation.'  
'This'
```

```
Pattern '\\A\\w+' (word at start of string)
```

```
'This is some text -- with punctuation.'  
'This'
```

```
Pattern '\\w+\\S*$' (word near end of string, skip punctuation)
```

```
'This is some text -- with punctuation.'  
.....'punctuation.'
```

```
Pattern '\\w+\\S*\\Z' (word near end of string, skip punctuation)
```

```
'This is some text -- with punctuation.'  
.....'punctuation.'
```

```
Pattern '\\w*t\\w*' (word containing t)
```

```
'This is some text -- with punctuation.'  
.....'text'  
.....'with'  
.....'punctuation'
```

```
Pattern '\\bt\\w+' (t at start of word)
```

```
'This is some text -- with punctuation.'  
.....'text'
```

```
Pattern '\\w+t\\b' (t at end of word)
```

```
'This is some text -- with punctuation.'  
.....'text'
```

```
Pattern '\\Bt\\B' (t, not start or end of word)
```

```
'This is some text -- with punctuation.'  
.....'t'  
.....'t'  
.....'t'
```



1.3.5 限制搜索

如果提前已经知道只需搜索整个输入的一个子集，可以告诉 `re` 限制搜索范围，从而进一步约束正则表达式匹配。例如，如果模式必须出现在输入的最前面，那么使用 `match()` 而不是 `search()` 会锚定搜索，而不必在搜索模式中显式地包含一个锚。

```
import re

text = 'This is some text -- with punctuation.'
pattern = 'is'
print 'Text :', text
print 'Pattern:', pattern

m = re.match(pattern, text)
print 'Match :', m
s = re.search(pattern, text)
print 'Search :', s
```

由于字面量文本 `is` 未出现在输入文本最前面，因此使用 `match()` 无法找到它。不过，这个序列在文本中另外出现了两次，所以 `search()` 可以找到它。

```
$ python re_match.py

Text : This is some text -- with punctuation.
Pattern: is
Match : None
Search : <_sre.SRE_Match object at 0x100d2bed0>
```

已编译正则表达式的 `search()` 方法还接受可选的 `start` 和 `end` 位置参数，将搜索限制在输入的一个子串中。

```
import re

text = 'This is some text -- with punctuation.'
pattern = re.compile(r'\b\w*is\w*\b')

print 'Text:', text
print

pos = 0
while True:
    match = pattern.search(text, pos)
    if not match:
        break
    s = match.start()
    e = match.end()
    print ' %2d : %2d = "%s" % \
        (s, e-1, text[s:e])
```



```
# Move forward in text for the next search
pos = e
```

这个例子实现了 `iterall()` 的一种不太高效的形式。每次找到一个匹配时，该匹配的结束位置将用于下一次搜索。

```
$ python re_search_substring.py
```

```
Text: This is some text -- with punctuation.
```

```
0 : 3 = "This"
5 : 6 = "is"
```

1.3.6 用组解析匹配

搜索模式匹配是正则表达式所提供强大功能的基础。为模式增加组 (group) 可以隔离匹配文本的各个部分，进一步扩展这些功能来创建一个解析工具。通过将模式包围在小括号中 (即“(”和“)”) 来分组。

```
from re_test_patterns import test_patterns

test_patterns(
    'abbbaabbbbbaaaaa',
    [ ('a(ab)', 'a followed by literal ab'),
      ('a(a*b*)', 'a followed by 0-n a and 0-n b'),
      ('a(ab)*', 'a followed by 0-n ab'),
      ('a(ab)+', 'a followed by 1-n ab'),
    ])
```

任何完整的正则表达式都可以转换为组，并嵌套在一个更大的表达式中。所有重复修饰符可以应用到整个组作为一个整体，这就要求重复整个组模式。

```
$ python re_groups.py
```

```
Pattern 'a(ab)' (a followed by literal ab)
```

```
'abbbaabbbbbaaaaa'
....'aab'
```

```
Pattern 'a(a*b*)' (a followed by 0-n a and 0-n b)
```

```
'abbbaabbbbbaaaaa'
'abb'
...'aaabbbb'
.....'aaaaa'
```

```
Pattern 'a(ab)*' (a followed by 0-n ab)
```

```
'abbbaabbbbbaaaaa'
```



```
'a'
... 'a'
.... 'aab'
..... 'a'
..... 'a'
..... 'a'
..... 'a'
..... 'a'
```

Pattern 'a(ab)+' (a followed by 1-n ab)

```
'abbbaabbbbbaaaaa'
.... 'aab'
```

要访问一个模式中单个组所匹配的子串，可以使用 Match 对象的 groups() 方法。

```
import re

text = 'This is some text -- with punctuation.'

print text
print

patterns = [
    (r'^(\w+)', 'word at start of string'),
    (r'(\w+)\S*$', 'word at end, with optional punctuation'),
    (r'(\bt\w+)\W+(\w+)', 'word starting with t, another word'),
    (r'(\w+t)\b', 'word ending with t'),
]

for pattern, desc in patterns:
    regex = re.compile(pattern)
    match = regex.search(text)
    print 'Pattern %r (%s)\n' % (pattern, desc)
print ' ', match.groups()
print
```

Match.groups() 会按表达式中与字符串匹配的组的顺序返回一个字符串序列。

```
$ python re_groups_match.py
```

```
This is some text -- with punctuation.
```

```
Pattern '^(\w+)' (word at start of string)
```

```
('This',)
```

```
Pattern '(\w+)\S*$' (word at end, with optional punctuation)
```

```
('punctuation',)
```



```

Pattern '(\bt\w+)\W+(\w+)' (word starting with t, another word)

('text', 'with')

Pattern '(\w+t)\b' (word ending with t)

('text',)

```

使用 `group()` 可以得到某个组的匹配。如果使用分组来查找字符串的各部分，不过结果中并不需要某些与组匹配的部分，此时 `group()` 会很有用。

```

import re

text = 'This is some text -- with punctuation.'

print 'Input text          :', text

# word starting with 't' then another word
regex = re.compile(r'(\bt\w+)\W+(\w+)')
print 'Pattern            :', regex.pattern

match = regex.search(text)
print 'Entire match       :', match.group(0)
print 'Word starting with "t":', match.group(1)
print 'Word after "t" word :', match.group(2)

```

第 0 组表示与整个表达式匹配的字符串，子组按其左小括号在表达式中出现的顺序从 1 开始标号。

```

$ python re_groups_individual.py

Input text          : This is some text -- with punctuation.
Pattern            : (\bt\w+)\W+(\w+)
Entire match       : text -- with
Word starting with "t": text
Word after "t" word : with

```

Python 对基本分组语法做了扩展，增加了命名组 (named group)。通过使用名字来指示组，这样以后就可以更容易地修改模式，而不必同时修改使用了匹配结果的代码。要设置一个组的名字，可以使用以下语法：(P<name>pattern)。

```

import re

text = 'This is some text -- with punctuation.'

print text
print

```

```

for pattern in [ r'^(?P<first_word>\w+)',
                 r'(?P<last_word>\w+)\S*$',
                 r'(?P<t_word>\bt\w+)\W+(?P<other_word>\w+)',
                 r'(?P<ends_with_t>\w+t)\b',
                 ]:
    regex = re.compile(pattern)
    match = regex.search(text)
    print 'Matching "%s"' % pattern
    print ' ', match.groups()
    print ' ', match.groupdict()
    print

```

使用 `groupdict()` 可以获取一个字典，它将组名映射到匹配的子串。`groups()` 返回的有序序列还包含命名模式。

```
$ python re_groups_named.py
```

```
This is some text -- with punctuation.
```

```

Matching "^(?P<first_word>\w+)"
('This',)
{'first_word': 'This'}

```

```

Matching "(?P<last_word>\w+)\S*$"
('punctuation',)
{'last_word': 'punctuation'}

```

```

Matching "(?P<t_word>\bt\w+)\W+(?P<other_word>\w+)"
('text', 'with')
{'other_word': 'with', 't_word': 'text'}

```

```

Matching "(?P<ends_with_t>\w+t)\b"
('text',)
{'ends_with_t': 'text'}

```

以下是更新后的 `test_patterns()`，它会显示与一个模式匹配的编号组和命名组，使后面的例子更容易理解。

```

import re

def test_patterns(text, patterns=[]):
    """Given source text and a list of patterns, look for
    matches for each pattern within the text and print
    them to stdout.
    """
    # Look for each pattern in the text and print the results
    for pattern, desc in patterns:
        print 'Pattern %r (%s)\n' % (pattern, desc)
        print '  %r' % text

```

```

    for match in re.finditer(pattern, text):
        s = match.start()
        e = match.end()
        prefix = ' ' * (s)
        print ' %s%s' % (prefix, text[s:e], ' '*(len(text)-e)),
        print match.groups()
        if match.groupdict():
            print '%s%s' % (' ' * (len(text)-s), match.groupdict())
    print
    return

```

因为组本身也是一个完整的正则表达式，所以组可以嵌套在其他组中，构成更复杂的表达式。

```
from re_test_patterns_groups import test_patterns
```

```

test_patterns(
    'abbaabbba',
    [ (r'a((a*)(b*))', 'a followed by 0-n a and 0-n b'),
    ])

```

在这个例子中，组 (a*) 会匹配一个空串，所以 groups() 的返回值包括这个空串作为匹配值。

```
$ python re_groups_nested.py
```

```
Pattern 'a((a*)(b*))' (a followed by 0-n a and 0-n b)
```

```

'abbaabbba'
'abb'      ('bb', '', 'bb')
'aabbb'    ('abbb', 'a', 'bbb')
'a'        ('', '', '')

```

组对于指定候选模式也很有用。可以使用管道符号 (|) 指示应当匹配某一个或另一个模式。不过，要仔细考虑管道符号的放置。下面这个例子中的第一个表达式会匹配一个 a 序列后面跟有一个完全由某一个字母 (a 或 b) 构成的序列。第二个模式会匹配一个 a 后面跟有一个可能包含 a 或 b 的序列。这两个模式很相似，不过得到的匹配结果完全不同。

```
from re_test_patterns_groups import test_patterns
```

```

test_patterns(
    'abbaabbba',
    [ (r'a((a+)|(b+))', 'a then seq. of a or seq. of b'),
      (r'a((a|b)+)', 'a then seq. of [ab]'),
    ])

```

如果候选组不匹配，但是整个模式确实匹配，groups() 的返回值会在序列中本应出现候选组的位置上包含一个 None 值。

```
$ python re_groups_alternative.py
```

Pattern `'a((a+)|(b+))'` (a then seq. of a or seq. of b)

```
'abbaabbba'
'abb'      ('bb', None, 'bb')
'aa'      ('a', 'a', None)
```

Pattern `'a((a|b)+)'` (a then seq. of [ab])

```
'abbaabbba'
'abbaabbba' ('bbaabbba', 'a')
```

如果匹配子模式的字符串并不是从整个文本抽取的一部分，此时定义一个包含子模式的组也很有用。这些组称为“非捕获组”（noncapturing）。非捕获组可以用来描述重复模式或候选模式，而不在返回值中区分字符串的匹配部分。要创建一个非捕获组，可以使用语法 `(?:pattern)`。

```
from re_test_patterns_groups import test_patterns
```

```
test_patterns(
    'abbaabbba',
    [ (r'a((a+)|(b+))', 'capturing form'),
      (r'a(?:a+)|(?:b+)', 'noncapturing'),
    ])
```

对于一个模式，尽管其捕获和非捕获形式会匹配相同的结果，但是会返回不同的组，下面来加以比较。

```
$ python re_groups_noncapturing.py
```

Pattern `'a((a+)|(b+))'` (capturing form)

```
'abbaabbba'
'abb'      ('bb', None, 'bb')
'aa'      ('a', 'a', None)
```

Pattern `'a(?:a+)|(?:b+)'` (noncapturing)

```
'abbaabbba'
'abb'      ('bb',)
'aa'      ('a',)
```

1.3.7 搜索选项

利用选项标志可以改变匹配引擎处理表达式的方式。可以使用位或（OR）操作结合这些标志，然后传递至 `compile()`、`search()`、`match()` 以及其他接受匹配模式完成搜索的函数。

不区分大小写的匹配

IGNORECASE 使模式中的字面量字符和字符区间与大小写字符都匹配。

```

import re

text = 'This is some text -- with punctuation.'
pattern = r'\bT\w+'
with_case = re.compile(pattern)
without_case = re.compile(pattern, re.IGNORECASE)

print 'Text:\n %r' % text
print 'Pattern:\n %s' % pattern
print 'Case-sensitive:'
for match in with_case.findall(text):
    print ' %r' % match
print 'Case-insensitive:'
for match in without_case.findall(text):
    print ' %r' % match

```

由于这个模式包含字面量字符 T，但没有设置 IGNORECASE，因此只有一个匹配，即单词 This。如果忽略大小写，那么 text 也能匹配。

```
$ python re_flags_ignorecase.py
```

```

Text:
'This is some text -- with punctuation.'
Pattern:
\bT\w+
Case-sensitive:
'This'
Case-insensitive:
'This'
'text'

```

多行输入

有两个标志会影响如何在多行输入中进行搜索：MULTILINE 和 DOTALL。MULTILINE 标志会控制模式匹配代码如何对包含换行符的文本处理锚定指令。当打开多行模式时，除了整个字符串外，还要在每一行的开头和结尾应用 ^ 和 \$ 的锚定规则。

```

import re

text = 'This is some text -- with punctuation.\nA second line.'
pattern = r'^(^w+)|(\w+\S*$)'
single_line = re.compile(pattern)
multiline = re.compile(pattern, re.MULTILINE)

print 'Text:\n %r' % text
print 'Pattern:\n %s' % pattern
print 'Single Line : '
for match in single_line.findall(text):
    print ' %r' % (match,)

```

```

print 'Multiline      :'
for match in multiline.findall(text):
    print '  %r' % (match,)

```

这个例子中的模式会匹配输入的第一个或最后一个单词。它会匹配字符串末尾的 line., 尽管并没有换行符。

```
$ python re_flags_multiline.py
```

```

Text:
  'This is some text -- with punctuation.\nA second line.'
Pattern:
  (^\\w+)| (\\w+\\S*$)
Single Line :
  ('This', '')
  ('', 'line.')
Multiline   :
  ('This', '')
  ('', 'punctuation.')
  ('A', '')
  ('', 'line.')

```

DOTALL 也是一个与多行文本有关的标志。正常情况下, 点字符 (.) 可以与输入文本中除了换行符之外的所有其他字符匹配。这个标志则允许点字符还可以匹配合换行符。

```

import re

text = 'This is some text -- with punctuation.\nA second line.'
pattern = r'.+'
no_newlines = re.compile(pattern)
dotall = re.compile(pattern, re.DOTALL)

print 'Text:\n %r' % text
print 'Pattern:\n %s' % pattern
print 'No newlines : '
for match in no_newlines.findall(text):
    print '  %r' % match
print 'Dotall      : '
for match in dotall.findall(text):
    print '  %r' % match

```

如果没有这个标志, 输入文本的各行会与模式单独匹配。增加了这个标志后, 则会利用整个字符串。

```
$ python re_flags_dotall.py
```

```

Text:
  'This is some text -- with punctuation.\nA second line.'
Pattern:

```

```
.+
No newlines :
    'This is some text -- with punctuation.'
    'A second line.'
Dottall      :
    'This is some text -- with punctuation.\nA second line.'
```

Unicode

在 Python 2 中，str 对象使用的是 ASCII 字符集，而且正则表达式处理会假设模式和输入文本都是 ASCII 字符。先前描述的转义码就默认使用 ASCII 来定义。这些假设意味着模式 `\w+` 会匹配单词 “French” 但是不能匹配单词 “Fran ç ais”，这是因为 `ç` 不属于 ASCII 字符集。要在 Python 2 中启用 Unicode 匹配，需要在编译模式时或者调用模块级函数 `search()` 和 `match()` 时增加 Unicode 标志。

```
import re
import codecs
import sys

# Set standard output encoding to UTF-8.
sys.stdout = codecs.getwriter('UTF-8')(sys.stdout)

text = u'Français złoty Österreich'
pattern = ur'\w+'
ascii_pattern = re.compile(pattern)
unicode_pattern = re.compile(pattern, re.UNICODE)

print 'Text      :', text
print 'Pattern   :', pattern
print 'ASCII     :', u', '.join(ascii_pattern.findall(text))
print 'Unicode   :', u', '.join(unicode_pattern.findall(text))
```

对于 Unicode 文本，其他转义序列 (`\W`、`\b`、`\B`、`\d`、`\D`、`\s` 和 `\S`) 也会做不同的处理。正则表达式引擎不再假设字符集成员由转义序列标识，而会查看 Unicode 数据库来查找各个字符的属性。

```
$ python re_flags_unicode.py

Text      : Français złoty Österreich
Pattern   : \w+
ASCII     : Fran, ais, z, oty, sterreich
Unicode   : Français, złoty, Österreich
```

注意：Python 3 对所有字符串都默认使用 Unicode，所以这个标志已经不再需要。

详细表达式语法

随着表达式变得越来越复杂，紧凑格式的正则表达式语法可能会成为障碍。随着表达式

中组数的增加，需要做更多的工作来明确为什么需要各个元素以及表达式的各部分究竟如何交互。使用命名组有助于缓解这些问题，不过一种更好的解决方案是使用详细模式表达式 (verbose mode expression)，它允许在模式中嵌入注释和额外的空白符。

可以用一个验证 Email 地址的模式来说明采用详细模式能够更容易地处理正则表达式。第一个版本会识别以 3 个顶级域之一 (.com、.org 和 .edu) 结尾的地址。

```
import re

address = re.compile('[\w\d.+~]+@[([\w\d.]|\.)+(com/org/edu)',
                    re.UNICODE)

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com',
    u'valid-address@mail.example.com',
    u'not-valid@example.foo',
]

for candidate in candidates:
    match = address.search(candidate)
    print '%-30s %s' % (candidate, 'Matches' if match else 'No match')
```

这个表达式已经很复杂了，其中有多字符类、组和重复表达式。

```
$ python re_email_compact.py
```

```
first.last@example.com      Matches
first.last+category@gmail.com Matches
valid-address@mail.example.com Matches
not-valid@example.foo      No match
```

将这个表达式转换为一种更详细的格式，使之更容易扩展。

```
import re

address = re.compile(
    '''
    [\w\d.+~]+      # username
    @
    ([\w\d.]|\.)+   # domain name prefix
    (com/org/edu)   # TODO: support more top-level domains
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com',
    u'valid-address@mail.example.com',
```



```

    u'not-valid@example.foo',
]

for candidate in candidates:
    match = address.search(candidate)
    print '%-30s %s' % (candidate, 'Matches' if match else 'No match')

```

这个表达式会匹配同样的输入，不过采用这种扩展格式将更易读。注释还有助于标识模式中的不同部分，从而能扩展来匹配更多输入。

```
$ python re_email_verbose.py
```

```

first.last@example.com           Matches
first.last+category@gmail.com    Matches
valid-address@mail.example.com   Matches
not-valid@example.foo            No match

```

这个扩展的版本会解析包含人名和 Email 地址的输入（这很可能在 Email 首部出现）。首先是单独的人名，然后是 Email 地址，用尖括号包围（“<”和“>”）。

```

import re

address = re.compile(
    '''
    # A name is made up of letters, and may include "."
    # for title abbreviations and middle initials.
    ((?P<name>
        ([\w.,]+\s+)*([\w.,]+)
        \s*
        # Email addresses are wrapped in angle
        # brackets: < > but only if a name is
        # found, so keep the start bracket in this
        # group.
        <
    )? # the entire name is optional

    # The address itself: username@domain.tld
    (?P<email>
        ([\w\d.+~]+      # username
        @
        ([\w\d.]+"\.)+    # domain name prefix
        (com/org/edu)     # limit the allowed top-level domains
        )

    >? # optional closing angle bracket
    ''',
    re.UNICODE | re.VERBOSE)

```

```

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com',
    u'valid-address@mail.example.com',
    u'not-valid@example.foo',
    u'First Last <first.last@example.com>',
    u'No Brackets first.last@example.com',
    u'First Last',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
    u'<first.last@example.com>',
]

for candidate in candidates:
    print 'Candidate:', candidate
    match = address.search(candidate)
    if match:
        print '  Name :', match.groupdict()['name']
        print '  Email:', match.groupdict()['email']
    else:
        print '  No match'

```

类似于其他编程语言，能够在详细正则表达式中插入注释将有助于增强其可维护性。最后这个版本包含为将来的维护人员提供的实现说明，另外还包括一些空白符将各个组分开，并突出其嵌套层次。

```
$ python re_email_with_name.py
```

```

Candidate: first.last@example.com
  Name : None
  Email: first.last@example.com
Candidate: first.last+category@gmail.com
  Name : None
  Email: first.last+category@gmail.com
Candidate: valid-address@mail.example.com
  Name : None
  Email: valid-address@mail.example.com
Candidate: not-valid@example.foo
  No match
Candidate: First Last <first.last@example.com>
  Name : First Last
  Email: first.last@example.com
Candidate: No Brackets first.last@example.com
  Name : None
  Email: first.last@example.com
Candidate: First Last
  No match

```



```

Candidate: First Middle Last <first.last@example.com>
  Name : First Middle Last
  Email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
  Name : First M. Last
  Email: first.last@example.com
Candidate: <first.last@example.com>
  Name : None
  Email: first.last@example.com

```

在模式中嵌入标志

如果编译表达式时不能增加标志，如将模式作为参数传入一个将在以后编译该模式的库函数时，可以把标志嵌入到表达式字符串本身。例如，要启用不区分大小写的匹配，可以在表达式开头增加(?i)。

```

import re

text = 'This is some text -- with punctuation.'
pattern = r'(?i)\bT\w+'
regex = re.compile(pattern)

print 'Text      :', text
print 'Pattern   :', pattern
print 'Matches   :', regex.findall(text)

```

因为这些选项控制了如何计算或解析整个表达式，所以它们总要放在表达式最前面。

```

$ python re_flags_embedded.py

Text      : This is some text -- with punctuation.
Pattern   : (?i)\bT\w+
Matches   : ['This', 'text']

```

表 1.3 列出了所有标志的缩写。

表 1.3 正则表达式标志缩写

标 志	缩 写
IGNORECASE	i
MULTILINE	m
DOTALL	s
UNICODE	u
VERBOSE	x

可以把嵌入标志放在同一个组中结合使用。例如，(?imu)会打开相应选项，支持多行 Unicode 字符串不区分大小写的匹配。

1.3.8 前向或后向

很多情况下，仅当模式中另外某个部分也匹配时才匹配模式的某一部分，这可能很有用。例如，在 Email 解析表达式中，两个尖括号分别标志为可选。不过，实际上尖括号必须成对，只有当两个尖括号都出现或都不出现时表达式才能匹配。修改后的表达式使用了一个肯定前向 (positive look-ahead) 断言来匹配尖括号对。前向断言语法为 `(?=pattern)`。

```
import re

address = re.compile(
    '''
    # A name is made up of letters, and may include "."
    # for title abbreviations and middle initials.
    ((?P<name>
        ([\w.,]+\s+)*[\w.,,]+
    )
    \s+
    ) # name is no longer optional

    # LOOKAHEAD
    # Email addresses are wrapped in angle brackets, but only
    # if they are both present or neither is.
    (?(= (<.*>$)      # remainder wrapped in angle brackets
      |
      ([^<].*[^>]$) # remainder *not* wrapped in angle brackets
    )

    <? # optional opening angle bracket

    # The address itself: username@domain.tld
    (?P<email>
        [\w\d.-]+      # username
        @
        ([\w\d.]+\.)+   # domain name prefix
        (com/org/edu)   # limit the allowed top-level domains
    )

    >? # optional closing angle bracket
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'First Last <first.last@example.com>',
    u'No Brackets first.last@example.com',
    u'Open Bracket <first.last@example.com',
    u'Close Bracket first.last@example.com>',
]
```

```

for candidate in candidates:
    print 'Candidate:', candidate
    match = address.search(candidate)
    if match:
        print '  Name :', match.groupdict()['name']
        print '  Email:', match.groupdict()['email']
    else:
        print '  No match'

```

这个版本的表达式中出现了很多重要的变化。首先，`name` 部分不再是可选的。这说明，单独的地址将不能匹配，还能避免匹配那些格式不正确的“名/地址”组合。“`name`”组后面的肯定前向规则断言字符串的余下部分要么包围在一对尖括号中，要么不存在不匹配的尖括号；也就是尖括号要么都出现，要么都不出现。这个前向规则表述为一个组，不过前向组的匹配并不利用任何输入文本。这个模式的其余部分会从前向匹配之后的位置取字符。

```
$ python re_look_ahead.py
```

```

Candidate: First Last <first.last@example.com>
  Name : First Last
  Email: first.last@example.com
Candidate: No Brackets first.last@example.com
  Name : No Brackets
  Email: first.last@example.com
Candidate: Open Bracket <first.last@example.com
  No match
Candidate: Close Bracket first.last@example.com>
  No match

```

否定前向 (negative look-ahead) 断言 (`(?!pattern)`) 要求模式不匹配当前位置后面的文本。例如，Email 识别模式可以修改为忽略自动系统常用的 noreply 邮件地址。

```

import re

address = re.compile(
    '''
    ^
    # An address: username@domain.tld

    # Ignore noreply addresses
    (?!noreply@.*$)

    [\w\d.+~]+      # username
    @
    ([\w\d.]+\.)+    # domain name prefix
    (com/org/edu)    # limit the allowed top-level domains

    $
    ''',

```



```

re.UNICODE | re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'noreply@example.com',
]

for candidate in candidates:
    print 'Candidate:', candidate
    match = address.search(candidate)
    if match:
        print ' Match:', candidate[match.start():match.end()]
    else:
        print ' No match'

```

以 noreply 开头的地址与这个模式不匹配，因为前向断言失败。

```
$ python re_negative_look_ahead.py
```

```

Candidate: first.last@example.com
Match: first.last@example.com
Candidate: noreply@example.com
No match

```

不用前向检查 Email 地址 username 部分中的 noreply，还可以使用语法 (?<!pattern) 改写这个模式，写为在匹配 username 之后使用一个否定后向断言 (negative look-behind assertion)。

```

import re

address = re.compile(
    '''
    ^

    # An address: username@domain.tld

    [\w\d.+-]+          # username

    # Ignore noreply addresses
    (?<!noreply)

    @

    ([\w\d.]+\.)+        # domain name prefix
    (com|org|edu)         # limit the allowed top-level domains

    $
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [

```

```

    u'first.last@example.com',
    u'noreply@example.com',
]

for candidate in candidates:
    print 'Candidate:', candidate
    match = address.search(candidate)
    if match:
        print ' Match:', candidate[match.start():match.end()]
    else:
        print ' No match'

```

后向与前向匹配的做法稍有不同，表达式必须使用一个定长的模式。只要字符数固定（没有通配符或区间），后向匹配也允许重复。

```
$ python re_negative_look_behind.py
```

```

Candidate: first.last@example.com
Match: first.last@example.com
Candidate: noreply@example.com
No match

```

可以借助语法 (`?<=pattern`) 用肯定后向 (positive look-behind) 断言查找符合某个模式的文本。例如，以下表达式可以查找 Twitter handles。

```

import re

twitter = re.compile(
    '''
    # A twitter handle: @username
    (?<=@)
    ([\w\d_]+)      # username
    ''',
    re.UNICODE | re.VERBOSE)

text = '''This text includes two Twitter handles.
One for @ThePSF, and one for the author, @doughellmann.
'''

print text
for match in twitter.findall(text):
    print 'Handle:', match

```

这个模式会匹配能构成一个 Twitter 句柄的字符序列，只要字符序列前面有一个 @。

```
$ python re_look_behind.py
```

```

This text includes two Twitter handles.
One for @ThePSF, and one for the author, @doughellmann.

```

```
Handle: ThePSF
Handle: dougHELLmann
```

1.3.9 自引用表达式

匹配的值还可以用在表达式后面的部分中。例如，前面的 Email 例子可以更新为只匹配由人名和姓组成的地址，为此要包含这些组的反向引用（back-reference）。要达到这个目的，最容易的办法就是使用 `\num` 按 id 编号引用先前匹配的组。

```
import re

address = re.compile(
    r'''

    # The regular name
    (\w+)          # first name
    \s+
    (([\w.]+)\s+)? # optional middle name or initial
    (\w+)          # last name

    \s+

    <

    # The address: first_name.last_name@domain.tld
    (?P<email>
        \1          # first name
        \.
        \4          # last name
        @
        ([\w\d.]+\.)+ # domain name prefix
        (com|org|edu) # limit the allowed top-level domains
    )

    >
    ''',
    re.UNICODE | re.VERBOSE | re.IGNORECASE)

candidates = [
    u'First Last <first.last@example.com>',
    u'Different Name <first.last@example.com>',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
]

for candidate in candidates:
    print 'Candidate:', candidate
```



```

match = address.search(candidate)
if match:
    print ' Match name :', match.group(1), match.group(4)
    print ' Match email:', match.group(5)
else:
    print ' No match'

```

尽管这个语法很简单, 不过按数字 id 创建反向引用有两个缺点。从实用角度讲, 当表达式改变时, 这些组就必须重新编号, 每个引用可能都需要更新。另一个缺点是, 采用这种方法只能创建 99 个引用, 因为如果 id 编号有 3 位, 就会解释为一个八进制字符值而不是一个组引用。另一方面, 如果一个表达式有超过 99 个组, 问题就不只是无法引用表达式中的某些组那么简单了, 还会产生一些更严重的维护问题。

```

$ python re_refer_to_group.py

Candidate: First Last <first.last@example.com>
Match name : First Last
Match email: first.last@example.com
Candidate: Different Name <first.last@example.com>
No match
Candidate: First Middle Last <first.last@example.com>
Match name : First Last
Match email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
Match name : First Last
Match email: first.last@example.com

```

Python 的表达式解析器包括一个扩展, 可以使用 (?P=name) 指示表达式中先前匹配的一个命名组的值。

```

import re

address = re.compile(
    '''
    # The regular name
    (?P<first_name>\w+)
    \s+
    (([\w.]+)\s+)?      # optional middle name or initial
    (?P<last_name>\w+)

    \s+

    <

    # The address: first_name.last_name@domain.tld
    (?P<email>
        (?P=first_name)

```

```

\.(
    (?P=last_name)
    @
    ([\w\d.]+\.)+      # domain name prefix
    (com/org/edu)      # limit the allowed top-level domains
)

>
'''
re.UNICODE | re.VERBOSE | re.IGNORECASE)

candidates = [
    u'First Last <first.last@example.com>',
    u'Different Name <first.last@example.com>',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
]

for candidate in candidates:
    print 'Candidate:', candidate
    match = address.search(candidate)
    if match:
        print ' Match name :', match.groupdict()['first_name'],
        print match.groupdict()['last_name']
        print ' Match email:', match.groupdict()['email']
    else:
        print ' No match'

```

编译地址表达式时打开了 IGNORECASE 标志，因为尽管正确的名字通常首字母会大写，但 Email 地址往往不会大写首字母。

```

$ python re_refer_to_named_group.py

Candidate: First Last <first.last@example.com>
Match name : First Last
Match email: first.last@example.com
Candidate: Different Name <first.last@example.com>
No match
Candidate: First Middle Last <first.last@example.com>
Match name : First Last
Match email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
Match name : First Last
Match email: first.last@example.com

```

在表达式中使用反向引用还有一种机制，即根据前一个组是否匹配来选择不同的模式。可以修正这个 Email 模式，使得如果出现名字就需要有尖括号，不过如果只有 Email 地址本身就

不需要尖括号。查看一个组是否匹配的语法是 `(?(id)yes-expression|no-expression)`，这里 `id` 是组名或编号，`yes-expression` 是组有值时使用的模式，`no-expression` 则是组没有值时使用的模式。

```
import re

address = re.compile(
    '''
    # A name is made up of letters, and may include "."
    # for title abbreviations and middle initials.
    (?P<name>
        ([\w.]+\s+)*[\w.]+
    )?
    \s*

    # Email addresses are wrapped in angle brackets, but
    # only if a name is found.
    (? (name)
        # remainder wrapped in angle brackets because
        # there is a name
        (?P<brackets>(?(<.*>$)))
        |
        # remainder does not include angle brackets without name
        (?(^[^<].*[^>]$))
    )

    # Only look for a bracket if the look-ahead assertion
    # found both of them.
    (? (brackets) <| \s*)

    # The address itself: username@domain.tld
    (?P<email>
        [\w\d.+-]+      # username
        @
        ([\w\d.]+\.)+    # domain name prefix
        (com|org|edu)    # limit the allowed top-level domains
    )

    # Only look for a bracket if the look-ahead assertion
    # found both of them.
    (? (brackets) >| \s*)

    $
    ''',
    re.UNICODE | re.VERBOSE)
```

```

candidates = [
    u'First Last <first.last@example.com>',
    u'No Brackets first.last@example.com',
    u'Open Bracket <first.last@example.com',
    u'Close Bracket first.last@example.com>',
    u'no.brackets@example.com',
]

for candidate in candidates:
    print 'Candidate:', candidate
    match = address.search(candidate)
    if match:
        print '  Match name :', match.groupdict()['name']
        print '  Match email:', match.groupdict()['email']
    else:
        print '  No match'

```

这个版本的 Email 地址解析器使用了两个测试。如果 name 组匹配，则前向断言要求两个尖括号都出现，并建立 brackets 组。如果 name 不匹配，这个断言则要求余下文本不能用尖括号括起来。接下来，如果设置了 brackets 组，具体的模式匹配代码会借助字面量模式利用输入中的尖括号；否则，它会利用所有空格。

```
$ python re_id.py
```

```

Candidate: First Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: No Brackets first.last@example.com
  No match
Candidate: Open Bracket <first.last@example.com
  No match
Candidate: Close Bracket first.last@example.com>
  No match
Candidate: no.brackets@example.com
  Match name : None
  Match email: no.brackets@example.com

```

1.3.10 用模式修改字符串

除了搜索文本外，re 还支持使用正则表达式作为搜索机制来修改文本，而且替换可以引用正则表达式中的匹配组作为替换文本的一部分。使用 sub() 可以将一个模式的所有出现替换为另一个字符串。

```

import re

bold = re.compile(r'\{2}(.*?)\{2}')

```

```
text = 'Make this bold. This too.'
```

```
print 'Text:', text
print 'Bold:', bold.sub(r'<b>1</b>', text)
```

可以使用向后引用的 `\num` 语法插入与模式匹配的文本的引用。

```
$ python re_sub.py
```

```
Text: Make this bold. This too.
Bold: Make this <b>bold</b>. This <b>too</b>.
```

要在替换中使用命名组，可以使用语法 `\g<name>`。

```
import re
```

```
bold = re.compile(r'\{2}(?P<bold_text>.*?)\{2}', re.UNICODE)
```

```
text = 'Make this bold. This too.'
```

```
print 'Text:', text
print 'Bold:', bold.sub(r'<b>\g<bold_text></b>', text)
```

`\g<name>` 语法还适用于编号引用，使用这个语法可以消除组编号和两侧字面量数字之间的多义性。

```
$ python re_sub_named_groups.py
```

```
Text: Make this bold. This too.
Bold: Make this <b>bold</b>. This <b>too</b>.
```

向 `count` 传入一个值可以限制完成的替换数。

```
import re
```

```
bold = re.compile(r'\{2}(.*?)\{2}', re.UNICODE)
```

```
text = 'Make this bold. This too.'
```

```
print 'Text:', text
print 'Bold:', bold.sub(r'<b>1</b>', text, count=1)
```

由于 `count` 为 1，因此只完成了第一个替换。

```
$ python re_sub_count.py
```

```
Text: Make this bold. This too.
Bold: Make this <b>bold</b>. This too.
```

`subn()` 的工作原理与 `sub()` 很相似，只是它会同时返回修改后的字符串和完成的替换次数。

```
import re
```

```
bold = re.compile(r'\{2}(.*?)\{2}', re.UNICODE)
```

```
text = 'Make this bold. This too.'

print 'Text:', text
print 'Bold:', bold.subn(r'<b>\1</b>', text)
```

在这个例子中搜索模式有两次匹配。

```
$ python re_subn.py
Text: Make this bold. This too.
Bold: ('Make this <b>bold</b>. This <b>too</b>.', 2)
```

1.3.11 利用模式拆分

`str.split()` 是分解字符串来完成解析的最常用方法之一。不过，它只支持使用字面值作为分隔符。有时，如果输入没有一致的格式，就需要有一个正则表达式。例如，很多纯文本标记语言都把段落分隔符定义为两个或多个换行符 (`\n`)。在这种情况下，就不能使用 `str.split()`，因为这个定义中提到了“或多个”。

使用 `findall()` 标识段落有一种策略：使用类似 `(.+?)\n{2,}` 的模式。

```
import re

text = '''Paragraph one
on two lines.

Paragraph two.

Paragraph three.'''

for num, para in enumerate(re.findall(r'(.+?)\n{2,}',
                                     text,
                                     flags=re.DOTALL)
                           ):
    print num, repr(para)
    print
```

对于输入文本末尾的段落，这个模式会失败，原因在于“Paragraph three.”不是输出的一部分。

```
$ python re_paragraphs_findall.py

0 'Paragraph one\non two lines.'

1 'Paragraph two.'
```

可以扩展这个模式，指出段落以两个或更多个换行符结束或者以输入末尾作为结束，就能修正这个问题，但是会让模式更为复杂。可以转向使用 `re.split()` 而不是 `re.findall()`，这就能自

动地处理边界条件，并保证模式更简单。

```
import re

text = '''Paragraph one
on two lines.

Paragraph two.

Paragraph three.'''

print 'With findall:'
for num, para in enumerate(re.findall(r'(.+?) (\n{2,} |$)',
                                     text,
                                     flags=re.DOTALL)):
    print num, repr(para)
    print

print
print 'With split:'
for num, para in enumerate(re.split(r'\n{2,}', text)):
    print num, repr(para)
    print
```

`split()` 的模式参数更准确地表述了标记规范：由两个或更多个换行符标记输入字符串中段落之间的分隔点。

```
$ python re_split.py

With findall:
0 ('Paragraph one\non two lines.', '\n\n')

1 ('Paragraph two.', '\n\n\n')

2 ('Paragraph three.', '')
With split:
0 'Paragraph one\non two lines.'

1 'Paragraph two.'

2 'Paragraph three.'
```

可以将表达式包围在小括号里来定义一个组，这使得 `split()` 的工作方式更类似于 `str.partition()`，因此它会返回分隔符值以及字符串的其他部分。

```
import re
```

```
text = '''Paragraph one
on two lines.

Paragraph two.

Paragraph three.'''

print 'With split:'
for num, para in enumerate(re.split(r'(\n{2,})', text)):
    print num, repr(para)
    print
```

现在输出包括了各个段落，以及分隔这些段落的换行符序列。

```
$ python re_split_groups.py

With split:
0 'Paragraph one\non two lines.'

1 '\n\n'

2 'Paragraph two.'

3 '\n\n\n'

4 'Paragraph three.'
```

参见：

`re` (<http://docs.python.org/library/re.html>) 这个模块的标准库文档。

Regular Expression HOWTO (<http://docs.python.org/howto/regex.html>) Andrew Kuchling 为 Python 开发人员提供的正则表达式介绍。

Kodos (<http://kodos.sourceforge.net/>) 这是一个用于测试正则表达式的交互式工具，由 Phil Schwartz 创建。

Python Regular Expression Testing Tool (<http://www.pythonregex.com/>) 这是一个用来测试正则表达式的基于 Web 的工具，由 Brand Verity.com 的 David Naffziger 创建，灵感来自 Kodos。

Regular expression (http://en.wikipedia.org/wiki/Regular_expressions) 这是一篇维基百科文章，对正则表达式概念和技术做了一般介绍。

`locale` (15.2 节) 处理 Unicode 文本时可以使用 `locale` 模块设置语言配置。

`unicodedata` (docs.python.org/library/unicodedata.html) 通过程序访问 Unicode 字符属性数据库。

1.4 difflib——比较序列

作用：比较序列（特别是文本行）。

Python 版本：2.1 及以后版本

difflib 模块包含一些用来计算和处理序列之间差异的工具。它对于比较文本尤其有用，其中包含的函数可以使用多种常用差异格式生成报告。

本节中的例子都会使用 difflib_data.py 模块中以下这个常用的测试数据：

```
text1 = """Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
pulvinar porttitor tellus. Aliquam venenatis. Donec facilisis
pharetra tortor. In nec mauris eget magna consequat
convallis. Nam sed sem vitae odio pellentesque interdum. Sed
consequat viverra nisl. Suspendisse arcu metus, blandit quis,
rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate
tristique enim. Donec quis lectus a justo imperdiet tempus."""
text1_lines = text1.splitlines()

text2 = """Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
pulvinar, porttitor tellus. Aliquam venenatis. Donec facilisis
pharetra tortor. In nec mauris eget magna consequat
convallis. Nam cras vitae mi vitae odio pellentesque interdum. Sed
consequat viverra nisl. Suspendisse arcu metus, blandit quis,
rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Duis vulputate tristique enim. Donec quis lectus a
justo imperdiet tempus. Suspendisse eu lectus. In nunc."""

text2_lines = text2.splitlines()
```

1.4.1 比较文本体

Differ 类用于处理文本序列，并生成人类可读的差异（deltas）或更改指令，包括各行中的差异。Differ 生成的默认输出与 UNIX 下的 diff 命令行工具类似，包括两个列表的原始输入值（包含共同的值），以及指示做了哪些更改的标记数据。

- 有“-”前缀的行指示这些行在第一个序列中，但不包含在第二个序列中。
- 有“+”前缀的行在第二个序列中，但不包含在第一个序列中。
- 如果某一行在不同版本之间存在增量差异，会使用一个以“?”为前缀的额外的行强调新版本中的变更。

- 如果一行未改变，会输出该行，而且其左边有一个额外的空格，使它与其他可能有差异的输出对齐。

将文本传入 `compare()` 之前先分解为由单个文本行构成的序列，与传入大字符串相比，这样可以生成更可读的输出。

```
import difflib
from difflib_data import *

d = difflib.Differ()
diff = d.compare(text1_lines, text2_lines)
print '\n'.join(diff)
```

示例数据中两个文本段的开始部分是一样的，所以第一行会直接输出而没有任何额外标注。

```
Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
```

数据的第三行存在变化，修改后的文本中包含有一个逗号。这两个版本的数据行都会输出，而且第五行上的额外信息会显示文本中哪一列有修改，这里显示出增加了“,”字符。

```
- pulvinar porttitor tellus. Aliquam venenatis. Donec facilisis
+ pulvinar, porttitor tellus. Aliquam venenatis. Donec facilisis
?          +
```

输出中接下来几行显示删除了一个多余的空格。

```
- pharetra tortor. In nec mauris eget magna consequat
?
```

```
+ pharetra tortor. In nec mauris eget magna consequat
```

接下来有一个更为复杂的变更，替换了一个短语中的多个单词。

```
- convallis. Nam sed sem vitae odio pellentesque interdum. Sed
?          - -
```

```
+ convallis. Nam cras vitae mi vitae odio pellentesque interdum. Sed
?          +++ +++++ +
```

段落中下一句变化很大，所以表示差异时完全删除了老版本，而增加了新版本。

```
consequat viverra nisl. Suspendisse arcu metus, blandit quis,
rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
tristique vel, mauris. Curabitur vel lorem id nisl porta
- adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate
- tristique enim. Donec quis lectus a justo imperdiet tempus.
+ adipiscing. Duis vulputate tristique enim. Donec quis lectus a
+ justo imperdiet tempus. Suspendisse eu lectus. In nunc.
```

ndiff() 函数生成的输出基本上相同，会特别“加工”来处理文本数据，并删除输入中的“噪声”。

其他输出格式

Differ 类会显示所有输入行，统一差异格式 (unified diff) 则不同，它只包含已修改的文本行和一些上下文。Python 2.3 中增加了 unified_diff() 函数来生成这种输出。

```
import difflib
from difflib_data import *

diff = difflib.unified_diff(text1_lines,
                             text2_lines,
                             lineterm='',
                             )

print '\n'.join(list(diff))
```

lineterm 参数用来告诉 unified_diff() 不必为它返回的控制行追加换行符，因为输入行不包括这些换行符。输出时所有行都会增加换行符。对于 subversion 或其他版本控制工具的用户来说，输出看上去应该很熟悉。

```
$ python difflib_unified.py
```

```
---
+++
@@ -1,11 +1,11 @@
  Lorem ipsum dolor sit amet, consectetur adipiscing
  elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
  -pulvinar porttitor tellus. Aliquam venenatis. Donec facilisis
  -pharetra tortor. In nec mauris eget magna consequat
  -convallis. Nam sed sem vitae odio pellentesque interdum. Sed
  +pulvinar, porttitor tellus. Aliquam venenatis. Donec facilisis
  +pharetra tortor. In nec mauris eget magna consequat
  +convallis. Nam cras vitae mi vitae odio pellentesque interdum. Sed
  consequat viverra nisl. Suspendisse arcu metus, blandit quis,
  rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
  molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
  tristique vel, mauris. Curabitur vel lorem id nisl porta
  -adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate
  -tristique enim. Donec quis lectus a justo imperdiet tempus.
  +adipiscing. Duis vulputate tristique enim. Donec quis lectus a
  +justo imperdiet tempus. Suspendisse eu lectus. In nunc.
```

使用 context_diff() 会生成类似可读的输出。

1.4.2 无用数据

所有生成差异序列的函数都可以接受一些参数来指示应当忽略哪些行，以及应当忽略一行

中的哪些字符。例如，可以用这些参数指定跳过一个文件两个版本中的标记或空白符变更。

```
# This example is adapted from the source for difflib.py.
```

```
from difflib import SequenceMatcher

def show_results(s):
    i, j, k = s.find_longest_match(0, 5, 0, 9)
    print '  i = %d' % i
    print '  j = %d' % j
    print '  k = %d' % k
    print '  A[i:i+k] = %r' % A[i:i+k]
    print '  B[j:j+k] = %r' % B[j:j+k]

A = " abcd"
B = "abcd abcd"

print 'A = %r' % A
print 'B = %r' % B

print '\nWithout junk detection:'
show_results(SequenceMatcher(None, A, B))

print '\nTreat spaces as junk:'
show_results(SequenceMatcher(lambda x: x==" ", A, B))
```

默认情况下，Differ 不会显式忽略任何行或字符，而会依赖 SequenceMatcher 的能力检测噪声。ndiff() 的默认行为是忽略空格和制表符 (tab)。

```
$ python difflib_junk.py
A = ' abcd'
B = 'abcd abcd'
```

```
Without junk detection:
```

```
  i = 0
  j = 4
  k = 5
  A[i:i+k] = ' abcd'
  B[j:j+k] = ' abcd'
```

```
Treat spaces as junk:
```

```
  i = 1
  j = 0
  k = 4
  A[i:i+k] = 'abcd'
  B[j:j+k] = 'abcd'
```



1.4.3 比较任意类型

SequenceMatcher 类用于比较任意类型的两个序列，只要它们的值是可散列的。这个类使用一个算法来标识序列中最长的连续匹配块，并删除对实际数据没有贡献的无用值。

```
import difflib
from difflib_data import *

s1 = [ 1, 2, 3, 5, 6, 4 ]
s2 = [ 2, 3, 5, 4, 6, 1 ]

print 'Initial data:'
print 's1 =', s1
print 's2 =', s2
print 's1 == s2:', s1==s2
print

matcher = difflib.SequenceMatcher(None, s1, s2)
for tag, i1, i2, j1, j2 in reversed(matcher.get_opcodes()):

    if tag == 'delete':
        print 'Remove %s from positions [%d:%d]' % \
            (s1[i1:i2], i1, i2)
        del s1[i1:i2]
    elif tag == 'equal':
        print 's1[%d:%d] and s2[%d:%d] are the same' % \
            (i1, i2, j1, j2)

    elif tag == 'insert':
        print 'Insert %s from s2[%d:%d] into s1 at %d' % \
            (s2[j1:j2], j1, j2, i1)
        s1[i1:i2] = s2[j1:j2]

    elif tag == 'replace':
        print 'Replace %s from s1[%d:%d] with %s from s2[%d:%d]' % (
            s1[i1:i2], i1, i2, s2[j1:j2], j1, j2)
        s1[i1:i2] = s2[j1:j2]

print ' s1 =', s1

print 's1 == s2:', s1==s2
```

这个例子比较了两个整数列表，并使用 get_opcodes() 得出将原列表转换为新列表的指令。这里以逆序应用所做的修改，使得添加和删除元素之后列表索引仍是正确的。

```
$ python difflib_seq.py
```

```
Initial data:
```

```
s1 = [1, 2, 3, 5, 6, 4]
s2 = [2, 3, 5, 4, 6, 1]
s1 == s2: False
```

Replace [4] from s1[5:6] with [1] from s2[5:6]

```
s1 = [1, 2, 3, 5, 6, 1]
```

s1[4:5] and s2[4:5] are the same

```
s1 = [1, 2, 3, 5, 6, 1]
```

Insert [4] from s2[3:4] into s1 at 4

```
s1 = [1, 2, 3, 5, 4, 6, 1]
```

s1[1:4] and s2[0:3] are the same

```
s1 = [1, 2, 3, 5, 4, 6, 1]
```

Remove [1] from positions [0:1]

```
s1 = [2, 3, 5, 4, 6, 1]
```

```
s1 == s2: True
```

SequenceMatcher 用于处理定制类以及内置类型，前提是它们必须是可散列的。

参见：

difflib (<http://docs.python.org/library/difflib.html>) 这个模块的标准库文档。

Pattern Matching: The Gestalt Approach (<http://www.ddj.com/documents/s=1103/ddj8807c/>)

对 John W. Ratcliff 和 D. E. Metzener 提出的一种类似算法的讨论，发表于 1988 年 7 月的《Dr. Dobbs's Journal》。



第②章

数据结构

Python 包含很多标准编程数据结构，如 `list`（列表）、`tuple`（元组）、`dict`（字典）和 `set`（集合），这些都属于其内置类型。对很多应用来说这些结构已经足够，不再需要其他类型，不过，如果确实需要其他结构也大可放心，标准库提供了功能强大而且经过充分测试的版本可备使用。

`collections` 模块包含多种数据结构的实现，扩展了其他模块中的相应结构。例如，`Deque` 是一个双端队列，允许从任意一端增加或删除元素。`defaultdict` 是一个字典，如果找不到某个键，它会响应一个默认值，而 `OrderedDict` 会记住增加元素的序列。`namedtuple` 扩展了一般的 `tuple`，除了为每个成员元素提供一个数值索引外还提供一个属性名。

对于大量数据，`array` 会比 `list` 更高效地利用内存。由于 `array` 仅限于一种数据类型，与通用的 `list` 相比，它可以采用一种更紧凑的内存表示。不仅如此，`list` 的很多同样的方法都可以用来处理 `array`，所以可以把一个应用中的 `list` 替换为 `array` 而无须太多修改。

对一个序列中的元素排序是数据处理的一个基本方面。Python 的 `list` 包含一个 `sort()` 方法，不过有时维护一个有序列表会更为高效，而无须每次改变列表内容时都重新排序。`heapq` 中的函数可以修改列表的内容，同时还能以很低的开销维护列表原来的顺序。

构建有序列表或数组还有一种选择，即 `bisect`。它使用一种二分查找算法查找新元素的插入点，如果要反复对一个频繁改变的列表排序，可以将它作为一种候选方法。

尽管内置的 `list` 可以使用 `insert()` 和 `pop()` 方法模拟队列，但这不是线程安全的。要完成线程间的实序通信，可以使用 `Queue` 模块。`multiprocessing` 包含一个 `Queue` 的版本，它会处理进程间的通信，从而能更容易地将一个多线程程序转换为使用进程而不是线程。

`struct` 对于解码另一个应用的数据（可能来自一个二进制文件或数据流）会很有用，可以将这些数据解码为 Python 的内置类型，以便于处理。

本章会介绍两种与内存管理有关的模块。对于高度互连的数据结构，如图和树，可以使用 `weakref` 维护引用，同时当不再需要某些对象时仍允许垃圾回收器进行清理。`copy` 中的函数用于复制数据结构及其内容，包括用 `deepcopy()` 完成递归复制。

调试数据结构可能很耗费时间，特别是查看大序列或字典的打印输出。可以使用 `pprint` 创建易读的表示，从而打印到控制台或写至一个日志文件以利于调试。

另外，最后一点，如果现有的类型不能满足需求，可以派生某个内置类型进行定制，或者使用 `collections` 中定义的某个抽象基类作为起点构建一个新的容器类型。

2.1 collections——容器数据类型

作用：容器数据类型。

Python 版本：2.4 及以后版本

collections 模块包含除内置类型 list、dict 和 tuple 以外的其他容器数据类型。

2.1.1 Counter

Counter 作为一个容器，可以跟踪相同的值增加了多少次。这个类可以用来实现其他语言中常用包 (bag) 或多集合 (multiset) 数据结构来实现的算法。

初始化

Counter 支持 3 种形式的初始化。调用 Counter 的构造函数时可以提供一個元素序列或者一个包含键和计数的字典，还可以使用关键字参数将字符串名映射到计数。

```
import collections

print collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
print collections.Counter({'a':2, 'b':3, 'c':1})
print collections.Counter(a=2, b=3, c=1)
```

这 3 种形式的初始化结果都是一样的。

```
$ python collections_counter_init.py
```

```
Counter({'b': 3, 'a': 2, 'c': 1})
Counter({'b': 3, 'a': 2, 'c': 1})
Counter({'b': 3, 'a': 2, 'c': 1})
```

如果不提供任何参数，可以构造一个空 Counter，然后通过 update() 方法填充。

```
import collections

c = collections.Counter()
print 'Initial :', c

c.update('abcdaab')
print 'Sequence:', c

c.update({'a':1, 'd':5})
print 'Dict      :', c
```

计数值将根据新数据增加，替换数据不会改变计数。在下面的例子中，a 的计数会从 3 增加到 4。

```
$ python collections_counter_update.py
```

```
Initial : Counter()
Sequence: Counter({'a': 3, 'b': 2, 'c': 1, 'd': 1})
Dict      : Counter({'d': 6, 'a': 4, 'b': 2, 'c': 1})
```


访问计数

一旦填充了 Counter，可以使用字典 API 获取它的值。

```
import collections

c = collections.Counter('abcdaab')

for letter in 'abcde':
    print '%s : %d' % (letter, c[letter])
```

对于未知的元素，Counter 不会产生 KeyError。如果在输入中没有找到某个值（如此例中的 e），其计数为 0。

```
$ python collections_counter_get_values.py
```

```
a : 3
b : 2
c : 1
d : 1
e : 0
```

elements() 方法返回一个迭代器，将生成 Counter 知道的所有元素。

```
import collections

c = collections.Counter('extremely')
c['z'] = 0
print c
print list(c.elements())
```

不能保证元素的顺序不变，另外计数小于或等于 0 的元素不包含在内。

```
$ python collections_counter_elements.py
```

```
Counter({'e': 3, 'm': 1, 'l': 1, 'r': 1, 't': 1, 'y': 1, 'x': 1,
'z': 0})
['e', 'e', 'e', 'm', 'l', 'r', 't', 'y', 'x']
```

使用 most_common() 可以生成一个序列，其中包含 n 个最常遇到的输入值及其相应计数。

```
import collections

c = collections.Counter()
with open('/usr/share/dict/words', 'rt') as f:
    for line in f:
        c.update(line.rstrip().lower())

print 'Most common:'
for letter, count in c.most_common(3):
    print '%s: %7d' % (letter, count)
```

这个例子要统计系统字典的所有单词中出现的字母，来生成一个频度分布，然后打印 3 个最常见的字母。如果不向 `most_common()` 提供参数，会生成由所有元素构成的一个列表，按频度排序。

```
$ python collections_counter_most_common.py
```

```
Most common:
```

```
e: 234803
```

```
i: 200613
```

```
a: 198938
```

算术操作

`Counter` 实例支持算术和集合操作来完成结果的聚集。

```
import collections
```

```
c1 = collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
```

```
c2 = collections.Counter('alphabet')
```

```
print 'C1:', c1
```

```
print 'C2:', c2
```

```
print '\nCombined counts:'
```

```
print c1 + c2
```

```
print '\nSubtraction:'
```

```
print c1 - c2
```

```
print '\nIntersection (taking positive minimums):'
```

```
print c1 & c2
```

```
print '\nUnion (taking maximums):'
```

```
print c1 | c2
```

每次通过一个操作生成一个新的 `Counter` 时，计数为 0 或负数的元素都会被删除。在 `c1` 和 `c2` 中 `a` 的计数相同，所以减法操作后它的计数为 0。

```
$ python collections_counter_arithmetic.py
```

```
C1: Counter({'b': 3, 'a': 2, 'c': 1})
```

```
C2: Counter({'a': 2, 'b': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1, 't': 1})
```

```
Combined counts:
```

```
Counter({'a': 4, 'b': 4, 'c': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1, 't': 1})
```

```
Subtraction:
```

```
Counter({'b': 2, 'c': 1})
```

```

Intersection (taking positive minimums):
Counter({'a': 2, 'b': 1})

Union (taking maximums):
Counter({'b': 3, 'a': 2, 'c': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1,
        't': 1})

```

2.1.2 defaultdict

标准字典包括一个方法 `setdefault()` 来获取一个值，如果这个值不存在则建立一个默认值。与之相反，`defaultdict` 初始化容器时会让调用者提前指定默认值。

```

import collections

def default_factory():
    return 'default value'

d = collections.defaultdict(default_factory, foo='bar')
print 'd:', d
print 'foo =>', d['foo']
print 'bar =>', d['bar']

```

只要所有键都有相同的默认值并无不妥，就可以使用这个方法。如果默认值是一种用于聚集或累加值的类型，如 `list`、`set` 或者甚至是 `int`，这个方法尤其有用。标准库文档提供了很多采用这种方式使用 `defaultdict` 的例子。

```

$ python collections_defaultdict.py

d: defaultdict(<function default_factory
    at 0x100d9ba28>, {'foo': 'bar'})
foo => bar
bar => default value

```

参见：

`defaultdict examples` (<http://docs.python.org/lib/defaultdict-examples.html>) 标准库文档中使用 `defaultdict` 的例子。

`Evolution of Default Dictionaries in Python` (http://jtauber.com/blog/2008/02/27/evolution_of_default_dictionaries_in_python/) James Tauber 对 `defaultdict` 与初始化字典的其他方式之间的关联所做的讨论。

2.1.3 deque

`deque`（双端队列）支持从任意一端增加和删除元素。更为常用的两种结构，即栈和队列，就是双端队列的退化形式，其输入和输出限制在一端。

```

import collections

```

```
d = collections.deque('abcdefg')
print 'Deque:', d
print 'Length:', len(d)
print 'Left end:', d[0]
print 'Right end:', d[-1]

d.remove('c')
print 'remove(c):', d
```

由于 deque 是一种序列容器，因此同样支持 list 的一些操作，如用 `__getitem__()` 检查内容、确定长度，以及通过匹配标识从序列中间删除元素。

```
$ python collections_deque.py

Deque: deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
Length: 7
Left end: a
Right end: g
remove(c): deque(['a', 'b', 'd', 'e', 'f', 'g'])
```

填充

deque 可以从任意一端填充，在 Python 实现中称为“左端”和“右端”。

```
import collections
```

```
# Add to the right
d1 = collections.deque()
d1.extend('abcdefg')
print 'extend      :', d1
d1.append('h')
print 'append      :', d1

# Add to the left
d2 = collections.deque()
d2.extendleft(xrange(6))
print 'extendleft:', d2
d2.appendleft(6)
print 'appendleft:', d2
```

`extendleft()` 函数迭代处理其输入，对各个元素完成与 `appendleft()` 同样的处理。最终结果是 deque 将包含逆序的输入序列。

```
$ python collections_deque_populating.py

extend      : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
append      : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
extendleft: deque([5, 4, 3, 2, 1, 0])
appendleft: deque([6, 5, 4, 3, 2, 1, 0])
```

利用

类似地，可以从两端或任意一端利用 deque 的元素，这取决于所应用的算法。

```
import collections

print 'From the right:'
d = collections.deque('abcdefg')
while True:
    try:
        print d.pop(),
    except IndexError:
        break
print

print '\nFrom the left:'
d = collections.deque(xrange(6))
while True:
    try:
        print d.popleft(),
    except IndexError:
        break
print
```

使用 pop() 可以从 deque 的右端删除一个元素，使用 popleft() 可以从 deque 的左端删除一个元素。

```
$ python collections_deque_consuming.py
```

```
From the right:
g f e d c b a
```

```
From the left:
0 1 2 3 4 5
```

由于双端队列是线程安全的，所以甚至可以在不同线程中同时从两端利用队列的内容。

```
import collections
import threading
import time

candle = collections.deque(xrange(5))

def burn(direction, nextSource):
    while True:
        try:
            next = nextSource()
        except IndexError:
            break
        else:
            print '%8s: %s' % (direction, next)
```

```

        time.sleep(0.1)
    print '%8s done' % direction
    return

left = threading.Thread(target=burn, args=('Left', candle.popleft))
right = threading.Thread(target=burn, args=('Right', candle.pop))

left.start()
right.start()

left.join()
right.join()

```

这个例子中的线程交替处理两端，删除元素，直至这个 deque 为空。

```
$ python collections_deque_both_ends.py
```

```

Left: 0
Right: 4
Right: 3
Left: 1
Right: 2
Left done
Right done

```

旋转

deque 的另一个很有用的功能是可以按任意一个方向旋转，而跳过一些元素。

```

import collections

d = collections.deque(xrange(10))
print 'Normal      : ', d

d = collections.deque(xrange(10))
d.rotate(2)
print 'Right rotation:', d

d = collections.deque(xrange(10))
d.rotate(-2)
print 'Left rotation : ', d

```

将 deque 向右旋转（使用一个正旋转值），会从右端取元素，把它们移到左端。向左旋转（使用一个负旋转值）则从左端将元素移至右端。可以形象地把 deque 中的元素看作刻在拨号盘上，这对于理解双端队列很有帮助。

```
$ python collections_deque_rotate.py
```

```

Normal      : deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Right rotation: deque([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
Left rotation : deque([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])

```

参见:

Deque (<http://en.wikipedia.org/wiki/Deque>) 维基百科文章, 提供了对双端队列数据结构的讨论。

Deque Recipes (<http://docs.python.org/lib/deque-recipes.html>) 标准库文档的算法中使用双端队列的例子。

2.1.4 namedtuple

标准 tuple 使用数值索引来访问其成员。

```
bob = ('Bob', 30, 'male')
print 'Representation:', bob

jane = ('Jane', 29, 'female')
print '\nField by index:', jane[0]

print '\nFields by index:'
for p in [ bob, jane ]:
    print '%s is a %d year old %s' % p
```

因此对于简单用途来说, tuple 是很方便的容器。

```
$ python collections_tuple.py
```

```
Representation: ('Bob', 30, 'male')
```

```
Field by index: Jane
```

```
Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female
```

另一方面, 使用 tuple 时需要记住对应各个值要使用哪个索引, 这可能会导致错误, 特别是当 tuple 有大量字段, 而且元组的构造和使用相距很远时。对于各个成员, namedtuple 除了指定数值索引外, 还会指定名字。

定义

namedtuple 实例与常规元组在内存使用方面同样高效, 因为它们没有各实例的字典。各种 namedtuple 都由其自己的类表示, 使用 namedtuple() 工厂函数来创建。参数就是新类名和一个包含元素名的字符串。

```
import collections

Person = collections.namedtuple('Person', 'name age gender')

print 'Type of Person:', type(Person)

bob = Person(name='Bob', age=30, gender='male')
print '\nRepresentation:', bob
```

```
jane = Person(name='Jane', age=29, gender='female')
print '\nField by name:', jane.name

print '\nFields by index:'
for p in [ bob, jane ]:
    print '%s is a %d year old %s' % p
```

如这个例子所示，除了使用标准元组的位置索引外，还可以使用点记法 (obj.attr) 按名字访问 `namedtuple` 的字段。

```
$ python collections_namedtuple_person.py

Type of Person: <type 'type'>

Representation: Person(name='Bob', age=30, gender='male')

Field by name: Jane

Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female
```

非法字段名

如果字段名重复或与 Python 关键字冲突，就是非法字段名。

```
import collections

try:
    collections.namedtuple('Person', 'name class age gender')
except ValueError, err:
    print err

try:
    collections.namedtuple('Person', 'name age gender age')
except ValueError, err:
    print err
```

解析字段名时，非法值会导致 `ValueError` 异常。

```
$ python collections_namedtuple_bad_fields.py
```

```
Type names and field names cannot be a keyword: 'class'
Encountered duplicate field name: 'age'
```

如果创建一个 `namedtuple` 时要基于在程序控制之外的值（如表示一个数据库查询返回的记录行，而且数据库模式事先并不知道），要将 `rename` 选项设置为 `True`，从而对非法字段重命名。

```
import collections

with_class = collections.namedtuple(
```



```

    'Person', 'name class age gender',
    rename=True)
print with_class._fields

```

```

two_ages = collections.namedtuple(
    'Person', 'name age gender age',
    rename=True)
print two_ages._fields

```

重命名的字段的新名字取决于它在 tuple 中的索引，所以名为 class 的字段会变成 _1，重复的 age 字段则变成 _3。

```
$ python collections_namedtuple_rename.py
```

```

('name', '_1', 'age', 'gender')
('name', 'age', 'gender', '_3')

```

2.1.5 OrderedDict

OrderedDict 是一个字典子类，可以记住其内容增加的顺序。

```
import collections
```

```
print 'Regular dictionary:'
```

```

d = {}
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'

```

```

for k, v in d.items():
    print k, v

```

```
print '\nOrderedDict:'
```

```

d = collections.OrderedDict()
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'

```

```

for k, v in d.items():
    print k, v

```

常规 dict 并不跟踪插入顺序，迭代处理时会根据键在散列表中存储的顺序来生成值。在 OrderedDict 中则相反，它会记住元素插入的顺序，并在创建迭代器时使用这个顺序。

```
$ python collections_orderreddict_iter.py
```

```
Regular dictionary:
```

```

a A
c C
b B

```

```
OrderedDict:
```

```
a A
b B
c C
```

相等性

常规的 dict 在检查相等性时会查看其内容。OrderedDict 还会考虑元素增加的顺序。

```
import collections
```

```
print 'dict'
```

```
d1 = {}
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'
```

```
d2 = {}
d2['c'] = 'C'
d2['b'] = 'B'
d2['a'] = 'A'
```

```
print d1 == d2
```

```
print 'OrderedDict:',
```

```
d1 = collections.OrderedDict()
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'
d2 = collections.OrderedDict()
d2['c'] = 'C'
d2['b'] = 'B'
d2['a'] = 'A'
```

```
print d1 == d2
```

在这个例子中，由于两个有序字典由不同顺序的值创建，所以认为这两个有序字典是不同的。

```
$ python collections_orderreddict_equality.py
```

```
dict          : True
OrderedDict: False
```

参见：

`collections` (<http://docs.python.org/library/collections.html>) 这个模块的标准库文档。

2.2 array——固定类型数据序列

作用：高效管理固定类型数值数据的序列。

Python 版本: 1.4 及以后版本

array 模块定义了一个序列数据结构, 看起来与 list 非常相似, 只不过所有成员都必须是相同的基本类型。可以参考 array 的标准库文档全面了解目前支持的所有类型。

2.2.1 初始化

array 实例化时可以提供一个参数来描述允许哪种数据类型, 还可以有一个初始的数据序列存储在数组中。

```
import array
import binascii

s = 'This is the array.'
a = array.array('c', s)

print 'As string:', s
print 'As array :', a
print 'As hex   :', binascii.hexlify(a)
```

在这个例子中, 数组配置为包含一个字节序列, 用一个简单的字符串初始化。

```
$ python array_string.py
```

```
As string: This is the array.
As array : array('c', 'This is the array.')
As hex   : 54686973206973207468652061727261792e
```

2.2.2 处理数组

类似于其他的 Python 序列, 可以采用同样的方式扩展和处理 array。

```
import array
import pprint

a = array.array('i', xrange(3))
print 'Initial :', a

a.extend(xrange(3))
print 'Extended:', a

print 'Slice   :', a[2:5]

print 'Iterator:'
print list(enumerate(a))
```

目前支持的操作包括分片、迭代以及向末尾增加元素。

```
$ python array_sequence.py
```

```
Initial : array('i', [0, 1, 2])
```



```
Extended: array('i', [0, 1, 2, 0, 1, 2])
Slice   : array('i', [2, 0, 1])
Iterator:
[(0, 0), (1, 1), (2, 2), (3, 0), (4, 1), (5, 2)]
```

2.2.3 数组与文件

可以使用高效读 / 写文件的专用内置方法将数组的内容写入文件或从文件读入数组。

```
import array
import binascii
import tempfile

a = array.array('i', xrange(5))
print 'A1:', a

# Write the array of numbers to a temporary file
output = tempfile.NamedTemporaryFile()
a.tofile(output.file) # must pass an *actual* file
output.flush()

# Read the raw data
with open(output.name, 'rb') as input:
    raw_data = input.read()
    print 'Raw Contents:', binascii.hexlify(raw_data)

# Read the data into an array
input.seek(0)
a2 = array.array('i')
a2.fromfile(input, len(a))
print 'A2:', a2
```

这个例子展示了直接从二进制文件读取原始数据，将它读入一个新的数组，并把字节转换为适当的类型。

```
$ python array_file.py
```

```
A1: array('i', [0, 1, 2, 3, 4])
Raw Contents: 00000000010000000020000000030000000040000000
A2: array('i', [0, 1, 2, 3, 4])
```

2.2.4 候选字节顺序

如果数组中的数据没有采用固有的字节顺序，或者在发送到一个采用不同字节顺序的系统（或在网络上发送）之前需要交换顺序，可以由 Python 转换整个数组而无须迭代处理每一个元素。

```
import array
import binascii
```

```

def to_hex(a):
    chars_per_item = a.itemsize * 2 # 2 hex digits
    hex_version = binascii.hexlify(a)
    num_chunks = len(hex_version) / chars_per_item
    for i in xrange(num_chunks):
        start = i*chars_per_item
        end = start + chars_per_item
        yield hex_version[start:end]

a1 = array.array('i', xrange(5))
a2 = array.array('i', xrange(5))
a2.byteswap()

fmt = '%10s %10s %10s %10s'
print fmt % ('A1 hex', 'A1', 'A2 hex', 'A2')
print fmt % (('-' * 10,) * 4)
for values in zip(to_hex(a1), a1, to_hex(a2), a2):
    print fmt % values

```

byteswap() 方法会交换 C 数组中元素的字节顺序, 比在 Python 中循环处理数据高效得多。

\$ python array_byteswap.py

A1 hex	A1	A2 hex	A2
00000000	0	00000000	0
01000000	1	00000001	16777216
02000000	2	00000002	33554432
03000000	3	00000003	50331648
04000000	4	00000004	67108864

参见:

array (<http://docs.python.org/library/array.html>) 这个模块的标准库文档。

struct (2.6 节) struct 模块。

Numerical Python (www.scipy.org) NumPy 是一个高效处理大数据集的 Python 库。

2.3 heapq——堆排序算法

作用: heapq 模块实现了一个适用于 Python 列表的最小堆排序算法。

Python 版本: 2.3 版本中新增, 并在 2.5 版本中做了补充

堆 (heap) 是一个树形数据结构, 其中子节点与父节点是一种有序关系。二叉堆 (Binary heap) 可以使用以如下方式组织的列表或数组表示, 即元素 N 的子元素位于 $2*N+1$ 和 $2*N+2$ (索引从 0 开始)。这种布局允许原地重新组织堆, 从而不必在增加或删除元素时分配大量内存。

最大堆 (max-heap) 确保父节点大于或等于其两个子节点。最小堆 (min-heap) 要求父节点小于或等于其子节点。Python 的 heapq 模块实现了一个最小堆。

2.3.1 示例数据

本节中的示例将使用 `heapq_heapdata.py` 中的数据。

```
# This data was generated with the random module.
```

```
data = [19, 9, 4, 10, 11]
```

堆输出使用 `heapq_showtree.py` 打印。

```
import math
from cStringIO import StringIO

def show_tree(tree, total_width=36, fill=' '):
    """Pretty-print a tree."""
    output = StringIO()
    last_row = -1
    for i, n in enumerate(tree):
        if i:
            row = int(math.floor(math.log(i+1, 2)))
        else:
            row = 0
        if row != last_row:
            output.write('\n')
        columns = 2**row
        col_width = int(math.floor((total_width * 1.0) / columns))
        output.write(str(n).center(col_width, fill))
        last_row = row
    print output.getvalue()
    print '-' * total_width
    print
    return
```

2.3.2 创建堆

创建堆有两种基本方式: `heappush()` 和 `heapify()`。

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

heap = []
print 'random :', data
print

for n in data:
    print 'add %3d:' % n
    heapq.heappush(heap, n)
    show_tree(heap)
```



使用 `heappush()` 时, 从数据源增加新元素时会保持元素的堆顺序。

```
$ python heapq_heappush.py
```

```
random : [19, 9, 4, 10, 11]
```

```
add 19:
```

```
          19
-----
```

```
add 9:
```

```
          9
        19
-----
```

```
add 4:
```

```
          4
        19          9
-----
```

```
add 10:
```

```
          4
        10          9
       19
-----
```

```
add 11:
```

```
          4
        10          9
       19      11
-----
```

如果数据已经在内存中, 使用 `heapify()` 原地重新组织列表中的元素会更高效。

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print 'random      :', data
heapq.heapify(data)
print 'heapified : '
show_tree(data)
```

如果按堆顺序一次一个元素地构建列表, 其结果与构建一个无序列表再调用 `heapify()` 是一

样的。

```
$ python heapq_heapify.py

random      : [19, 9, 4, 10, 11]
heapified :
```

```

          4
        9      19
      10      11
-----
```

2.3.3 访问堆的内容

一旦堆已正确组织，就可以使用 `heappop()` 删除有最小值的元素。

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data
print 'random      :', data
heapq.heapify(data)
print 'heapified :'
show_tree(data)
print

for i in xrange(2):
    smallest = heapq.heappop(data)
    print 'pop      %3d:' % smallest
    show_tree(data)
```

这个例子是由 `stdlib` 文档改写的，其中使用了 `heapify()` 和 `heappop()` 对一个数字列表排序。

```
$ python heapq_heappop.py

random      : [19, 9, 4, 10, 11]
heapified :
```

```

          4
        9      19
      10      11
-----
```

```
pop      4:

          9
        10      19
      11
-----
```




```

pop          9:

                10
            11      19
-----

```

如果希望在一个操作中删除现有元素并替换为新值，可以使用 `heapreplace()`。

```

import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data
heapq.heapify(data)
print 'start:'
show_tree(data)

for n in [0, 13]:
    smallest = heapq.heapreplace(data, n)
    print 'replace %2d with %2d:' % (smallest, n)
    show_tree(data)

```

通过原地替换元素，这样可以维持一个固定大小的堆，如按优先级排序的作业队列。

```
$ python heapq_heapreplace.py
```

```

start:

                4
            9      19
        10      11
-----

replace  4 with  0:

                0
            9      19
        10      11
-----

replace  0 with 13:

                9
            10      19
        13      11
-----

```

2.3.4 堆的数据极值

`heapq` 还包括两个检查可迭代对象的函数，查找其中包含的最大值或最小值的范围。

```
import heapq
from heapq import data
print 'all      :', data
print '3 largest :', heapq.nlargest(3, data)
print 'from sort :', list(reversed(sorted(data)[-3:]))
print '3 smallest:', heapq.nsmallest(3, data)
print 'from sort :', sorted(data)[:3]
```

只有当 n 值 ($n > 1$) 相对小时使用 `nlargest()` 和 `nsmallest()` 才算高效, 不过有些情况下这两个函数会很方便。

```
$ python heapq_extremes.py
```

```
all      : [19, 9, 4, 10, 11]
3 largest : [19, 11, 10]
from sort : [19, 11, 10]
3 smallest: [4, 9, 10]
from sort : [4, 9, 10]
```

参见:

`heapq` (<http://docs.python.org/library/heapq.html>) 这个模块的标准库文档。

`Heap (data structure)` ([http://en.wikipedia.org/wiki/Heap_\(data_structure\)](http://en.wikipedia.org/wiki/Heap_(data_structure))) 维基百科文章, 提供了对堆数据结构的一般描述。

2.5.3 节 基于标准库中 `Queue` (2.5 节) 的一个优先队列实现。

2.4 bisect——维护有序列表

作用: 维护有序列表, 而不必在每次向列表增加一个元素时都调用 `sort` 排序。

Python 版本: 1.4 及以后版本

`bisect` 模块实现了一个算法用于向列表中插入元素, 同时仍保持列表有序。有些情况下, 这比反复对一个列表排序更高效, 另外也比构建一个大列表之后再显式对其排序更为高效。

2.4.1 有序插入

下面给出一个简单的例子, 这里使用 `insert()` 按有序顺序向一个列表中插入元素。

```
import bisect
import random

# Use a constant seed to ensure that
# the same pseudo-random numbers
# are used each time the loop is run.
random.seed(1)

print 'New  Pos  Contents'
print '---  ---  -----'
```

```

# Generate random numbers and
# insert them into a list in sorted
# order.
l = []
for i in range(1, 15):
    r = random.randint(1, 100)
    position = bisect.bisect(l, r)
    bisect.insort(l, r)
    print '%3d %3d' % (r, position), l

```

输出的第一列显示了新随机数。第二列显示了这个数将插入到列表的哪个位置。每一行余下的部分则是当前的有序列表。

```
$ python bisect_example.py
```

New	Pos	Contents
14	0	[14]
85	1	[14, 85]
77	1	[14, 77, 85]
26	1	[14, 26, 77, 85]
50	2	[14, 26, 50, 77, 85]
45	2	[14, 26, 45, 50, 77, 85]
66	4	[14, 26, 45, 50, 66, 77, 85]
79	6	[14, 26, 45, 50, 66, 77, 79, 85]
10	0	[10, 14, 26, 45, 50, 66, 77, 79, 85]
3	0	[3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
84	9	[3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
44	4	[3, 10, 14, 26, 44, 45, 50, 66, 77, 79, 84, 85]
77	9	[3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
1	0	[1, 3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]

这是一个很简单的例子，对于此例所处理的数据量来说，如果直接构建列表然后完成一次排序，可能速度更快。不过对于长列表而言，使用类似这样的一个插入排序算法可以大大节省时间和内存。

2.4.2 处理重复

之前显示的结果集包括一个重复的值 77。bisect 模块提供了两种方法来处理重复。新值可以插入到现有值的左边或右边。insort() 函数实际上是 insort_right() 的别名，这个函数会在现有值之后插入新值。相应的函数 insort_left() 则在现有值之前插入新值。

```

import bisect
import random

# Reset the seed

```

```

random.seed(1)

print 'New Pos Contents'
print '--- --- -----'

# Use bisect_left and insort_left.
l = []
for i in range(1, 15):
    r = random.randint(1, 100)
    position = bisect.bisect_left(l, r)
    bisect.insort_left(l, r)
    print '%3d %3d' % (r, position), l

```

使用 `bisect_left()` 和 `insort_left()` 处理同样的数据时, 结果会得到相同的有序列表, 不过重复值插入的位置有所不同。

```
$ python bisect_example2.py
```

```

New Pos Contents
--- --- ----
14    0 [14]
85    1 [14, 85]
77    1 [14, 77, 85]
26    1 [14, 26, 77, 85]
50    2 [14, 26, 50, 77, 85]
45    2 [14, 26, 45, 50, 77, 85]
66    4 [14, 26, 45, 50, 66, 77, 85]
79    6 [14, 26, 45, 50, 66, 77, 79, 85]
10    0 [10, 14, 26, 45, 50, 66, 77, 79, 85]
3     0 [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
84    9 [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
44    4 [3, 10, 14, 26, 44, 45, 50, 66, 77, 79, 84, 85]
77    8 [3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
1     0 [1, 3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]

```

除了 Python 实现外, 还有一个速度更快的 C 实现。如果有 C 版本, 导入 `bisect` 时, 这个实现会自动地覆盖纯 Python 实现。

参见:

`bisect` (<http://docs.python.org/library/bisect.html>) 这个模块的标准库文档。

Insertion Sort (http://en.wikipedia.org/wiki/Insertion_sort) 维基百科文章, 提供了对插入排序算法的描述。

2.5 Queue——线程安全的 FIFO 实现

作用: 提供一个线程安全的 FIFO 实现。

Python 版本：至少 1.4

Queue 模块提供了一个适用于多线程编程的先进先出（first-in, first-out, FIFO）数据结构，可以用来在生产者和消费者线程之间安全地传递消息或其他数据。它会为调用者处理锁定，使多个线程可以安全地处理同一个 Queue 实例。Queue 的大小（其中包含的元素个数）可能要受限，以限制内存使用或处理。

注意：这里的讨论假设你已经了解队列的一般性质。如果你还不太清楚，在学习下面的内容之前可能需要先读一些有关的参考资料。

2.5.1 基本 FIFO 队列

Queue 类实现了一个基本的先进先出容器。使用 put() 将元素增加到序列一端，使用 get() 从另一端删除。

```
import Queue

q = Queue.Queue()

for i in range(5):
    q.put(i)

while not q.empty():
    print q.get(),
print
```

这个例子使用了一个线程，来展示按元素的插入顺序从队列删除元素。

```
$ python Queue_fifo.py
```

```
0 1 2 3 4
```

2.5.2 LIFO 队列

与 Queue 的标准 FIFO 实现相反，LifoQueue 使用了后进先出（last-in, first-out, LIFO）顺序（通常与栈数据结构关联）。

```
import Queue

q = Queue.LifoQueue()

for i in range(5):
    q.put(i)

while not q.empty():
    print q.get(),
print
```

get 将删除最近使用 put 插入到队列的元素。

```
$ python Queue_lifo.py
```

```
4 3 2 1 0
```

2.5.3 优先队列

有些情况下，队列中元素的处理顺序需要根据这些元素的特性来决定，而不只是在队列中创建或插入的顺序。例如，财务部门的打印作业可能要优先于一个开发人员打印的代码清单。PriorityQueue 使用队列内容的有序顺序来决定获取哪一个元素。

```
import Queue
import threading

class Job(object):
    def __init__(self, priority, description):
        self.priority = priority
        self.description = description
        print 'New job:', description
        return
    def __cmp__(self, other):
        return cmp(self.priority, other.priority)

q = Queue.PriorityQueue()

q.put( Job(3, 'Mid-level job') )
q.put( Job(10, 'Low-level job') )
q.put( Job(1, 'Important job') )

def process_job(q):
    while True:
        next_job = q.get()
        print 'Processing job:', next_job.description
        q.task_done()

workers = [ threading.Thread(target=process_job, args=(q,)),
            threading.Thread(target=process_job, args=(q,)),
            ]
for w in workers:
    w.setDaemon(True)
    w.start()

q.join()
```

这个例子有多个线程在处理作业，要根据调用 get() 时队列中元素的优先级来处理。运行消费者线程时，增加到队列中的元素的处理顺序取决于线程上下文切换。

```
$ python Queue_priority.py

New job: Mid-level job
New job: Low-level job
New job: Important job
Processing job: Important job
Processing job: Mid-level job
Processing job: Low-level job
```

2.5.4 构建一个多线程播客客户程序

本节将构建一个播客客户程序，程序的源代码展示了如何用多个线程使用 Queue 类。这个程序要读入一个或多个 RSS 提要，对专辑排队来显示最新的 5 集以供下载，并使用线程并行地处理多个下载。这里没有提供足够的错误处理，所以不能在实际生产环境中使用，不过这个骨架实现可以作为一个很好的例子来说明如何使用 Queue 模块。

首先要建立一些操作参数。正常情况下，这些参数来自用户输入（首选项、数据库，等等）。不过在这个例子中，线程数和要获取的 URL 列表都使用了硬编码值。

```
from Queue import Queue
from threading import Thread
import time
import urllib
import urlparse

import feedparser

# Set up some global variables
num_fetch_threads = 2
enclosure_queue = Queue()

# A real app wouldn't use hard-coded data...
feed_urls = [ 'http://advocacy.python.org/podcasts/littlebit.rss',
               ]
```

函数 downloadEnclosures() 在工作线程中运行，使用 urllib 来处理下载。

```
def downloadEnclosures(i, q):
    """This is the worker thread function.
    It processes items in the queue one after
    another. These daemon threads go into an
    infinite loop, and only exit when
    the main thread ends.
    """
    while True:
        print '%s: Looking for the next enclosure' % i
        url = q.get()
        parsed_url = urlparse.urlparse(url)
```

```

print '%s: Downloading:' % i, parsed_url.path
response = urllib.urlopen(url)
data = response.read()
# Save the downloaded file to the current directory
outfile_name = url.rpartition('/')[ -1]
with open(outfile_name, 'wb') as outfile:
    outfile.write(data)
q.task_done()

```

一旦定义了线程的目标函数，接下来可以启动工作线程。downloadEnclosures() 处理语句 url = q.get() 时，会阻塞并等待，直到队列返回某个结果。这说明，即使队列中没有任何内容，也可以安全地启动线程。

```

# Set up some threads to fetch the enclosures
for i in range(num_fetch_threads):
    worker = Thread(target=downloadEnclosures,
                    args=(i, enclosure_queue,))
    worker.setDaemon(True)
    worker.start()

```

下一步使用 Mark Pilgrim 的 feedparser 模块 (www.feedparser.org) 获取提要内容，并将这些专辑的 URL 入队。一旦第一个 URL 增加到队列，就会有某个工作线程提取这个 URL，开始下载。这个循环会继续增加元素，直到提要全部利用，工作线程会依次将 URL 出队来下载这些提要。

```

# Download the feed(s) and put the enclosure URLs into
# the queue.
for url in feed_urls:
    response = feedparser.parse(url, agent='fetch_podcasts.py')
    for entry in response['entries'][-5:]:
        for enclosure in entry.get('enclosures', []):
            parsed_url = urlparse.urlparse(enclosure['url'])
            print 'Queuing:', parsed_url.path
            enclosure_queue.put(enclosure['url'])

```

还有一件事要做，要使用 join() 再次等待队列腾空。

```

# Now wait for the queue to be empty, indicating that we have
# processed all the downloads.
print '*** Main thread waiting'
enclosure_queue.join()
print '*** Done'

```

运行这个示例脚本可以生成以下结果。

```
$ python fetch_podcasts.py
```

```

0: Looking for the next enclosure
1: Looking for the next enclosure
Queuing: /podcasts/littlebit/2010-04-18.mp3

```



```

Queuing: /podcasts/littlebit/2010-05-22.mp3
Queuing: /podcasts/littlebit/2010-06-06.mp3
Queuing: /podcasts/littlebit/2010-07-26.mp3
Queuing: /podcasts/littlebit/2010-11-25.mp3
*** Main thread waiting
0: Downloading: /podcasts/littlebit/2010-04-18.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-05-22.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-06-06.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-07-26.mp3
0: Looking for the next enclosure
0: Downloading: /podcasts/littlebit/2010-11-25.mp3
0: Looking for the next enclosure
*** Done

```

具体的输出取决于所使用的 RSS 提要的内容。

参见：

Queue (<http://docs.python.org/lib/module-Queue.html>) 这个模块的标准库文档。

2.1 节的 deque collections 模块包含一个 deque (双端队列) 类。

Queue data structures ([http://en.wikipedia.org/wiki/Queue_\(data_structure\)](http://en.wikipedia.org/wiki/Queue_(data_structure))) 解释队列的一篇维基百科文章。

FIFO (<http://en.wikipedia.org/wiki/FIFO>) 维基百科文章，解释了先进先出数据结构。

2.6 struct——二进制数据结构

作用：在字符串和二进制数据之间转换。

Python 版本：1.4 及以后版本

struct 模块包括一些在字节串与内置 Python 数据类型（如数字和字符串）之间完成转换的函数。

2.6.1 函数与 Struct 类

struct 提供了一组处理结构值的模块级函数，另外还有一个 Struct 类。格式指示符由字符串格式转换为一种编译表示，这与处理正则表达式的方式类似。这个转换会耗费资源，所以当创建一个 Struct 实例并在这个实例上调用方法时（而不是使用模块级函数），完成一次转换会更为高效。下面的例子都会使用 Struct 类。

2.6.2 打包和解包

Struct 支持使用格式指示符将数据打包 (packing) 为字符串，以及从字符串解包 (unpacking) 数据，格式提示符由表示数据类型的字符以及可选的数量及字节序 (endianness)

指示符构成。要全面了解目前支持的格式指示符，请参考标准库文档。

在下面的例子中，指示符要求有一个整数或 long 值、一个包含两字符的串，以及一个浮点数。格式指示符中包含的空格用来分隔类型指示符，在编译格式时会被忽略。

```
import struct
import binascii

values = (1, 'ab', 2.7)
s = struct.Struct('I 2s f')
packed_data = s.pack(*values)
print 'Original values:', values
print 'Format string  :', s.format
print 'Uses           :', s.size, 'bytes'
print 'Packed Value   :', binascii.hexlify(packed_data)
```

这个例子将打包的值转换为一个十六进制字节序列，以便利用 `binascii.hexlify()` 打印，因为有些字符是 null。

```
$ python struct_pack.py

Original values: (1, 'ab', 2.7)
Format string  : I 2s f
Uses           : 12 bytes
Packed Value   : 01000000061620000cdcc2c40
```

使用 `unpack()` 可以从打包表示中抽取数据。

```
import struct
import binascii

packed_data = binascii.unhexlify('01000000061620000cdcc2c40')

s = struct.Struct('I 2s f')
unpacked_data = s.unpack(packed_data)
print 'Unpacked Values:', unpacked_data
```

将打包值传入 `unpack()`，基本上会得到相同的值（注意浮点值中的差别）。

```
$ python struct_unpack.py

Unpacked Values: (1, 'ab', 2.700000047683716)
```

2.6.3 字节序

默认情况下，值会使用内置 C 库的字节序（endianness）来编码。只需在格式串中提供一个显式的字节序指令，就可以很容易地覆盖这个默认选择。

```
import struct
import binascii
```

```

values = (1, 'ab', 2.7)
print 'Original values:', values
endianness = [
    ('@', 'native, native'),
    ('=', 'native, standard'),
    ('<', 'little-endian'),
    ('>', 'big-endian'),
    ('!', 'network'),
]

for code, name in endianness:
    s = struct.Struct(code + ' I 2s f')
    packed_data = s.pack(*values)
    print
    print 'Format string :', s.format, 'for', name
    print 'Uses          :', s.size, 'bytes'
    print 'Packed Value   :', binascii.hexlify(packed_data)
    print 'Unpacked Value :', s.unpack(packed_data)

```

表 2.1 列出了 struct 使用的字节序指示符。

表 2.1 struct 的字节序指示符

指 示 符	含 义
@	内置顺序
=	内置标准
<	小端
>	大端
!	网络顺序

```
$ python struct_endianness.py
```

```
Original values: (1, 'ab', 2.7)
```

```
Format string : @ I 2s f for native, native
```

```
Uses          : 12 bytes
```

```
Packed Value   : 0100000061620000cdcc2c40
```

```
Unpacked Value : (1, 'ab', 2.700000047683716)
```

```
Format string : = I 2s f for native, standard
```

```
Uses          : 10 bytes
```

```
Packed Value   : 010000006162cdcc2c40
```

```
Unpacked Value : (1, 'ab', 2.700000047683716)
```

```
Format string : < I 2s f for little-endian
```

```
Uses          : 10 bytes
```



```

Packed Value : 0100000006162cdcc2c40
Unpacked Value : (1, 'ab', 2.700000047683716)

Format string : > I 2s f for big-endian
Uses : 10 bytes
Packed Value : 0000000016162402cccd
Unpacked Value : (1, 'ab', 2.700000047683716)

Format string : ! I 2s f for network
Uses : 10 bytes
Packed Value : 0000000016162402cccd
Unpacked Value : (1, 'ab', 2.700000047683716)

```

2.6.4 缓冲区

通常在重视性能的情况下或者向扩展模块传入或传出数据时才会处理二进制打包数据。通过避免为每个打包结构分配一个新缓冲区所带来的开销，可以优化这些情况。`pack_into()` 和 `unpack_from()` 方法支持直接写入预分配的缓冲区。

```

import struct
import binascii

s = struct.Struct('I 2s f')
values = (1, 'ab', 2.7)
print 'Original:', values

print
print 'ctypes string buffer'

import ctypes
b = ctypes.create_string_buffer(s.size)
print 'Before :', binascii.hexlify(b.raw)
s.pack_into(b, 0, *values)
print 'After :', binascii.hexlify(b.raw)
print 'Unpacked:', s.unpack_from(b, 0)
print
print 'array'

import array
a = array.array('c', '\0' * s.size)
print 'Before :', binascii.hexlify(a)
s.pack_into(a, 0, *values)
print 'After :', binascii.hexlify(a)
print 'Unpacked:', s.unpack_from(a, 0)

```

Struct 的 `size` 属性指出缓冲区需要有多大。

```
$ python struct_buffers.py
```



```
Original: (1, 'ab', 2.7)

ctypes string buffer
Before : 000000000000000000000000
After  : 0100000061620000cdcc2c40
Unpacked: (1, 'ab', 2.700000047683716)
```

```
array
Before : 000000000000000000000000
After  : 0100000061620000cdcc2c40
Unpacked: (1, 'ab', 2.700000047683716)
```

参见:

struct (<http://docs.python.org/library/struct.html>) 这个模块的标准库文档。

array(2.2 节) array 模块, 用于处理固定类型值的序列。

binascii (<http://docs.python.org/library/binascii.html>) binascii 模块, 用于生成二进制数据的 ASCII 表示。

Endianness (<http://en.wikipedia.org/wiki/Endianness>) 维基百科文章, 提供了字节顺序以及编码中字节序的解释。

2.7 weakref——对象的非永久引用

作用: 引用一个“昂贵”的对象, 不过如果不再有其他非弱引用, 则允许由垃圾回收器回收其内存。

Python 版本: 2.1 及以后版本

weakref 模块支持对象的弱引用。正常的引用会增加对象的引用计数, 避免它被垃圾回收。但并不总希望如此, 比如有时可能会出现一个循环引用, 或者有时可能要构建一个对象缓存, 需要内存时则要删除这个缓存。弱引用 (weak reference) 是避免对象被自动清除的一个对象句柄。

2.7.1 引用

对象的弱引用通过 ref 类管理。要获取原对象, 可以调用引用对象。

```
import weakref

class ExpensiveObject(object):
    def __del__(self):
        print '(Deleting %s)' % self

obj = ExpensiveObject()
r = weakref.ref(obj)

print 'obj:', obj
```

```

print 'ref:', r
print 'r():', r()

print 'deleting obj'
del obj
print 'r():', r()

```

在这里，由于 obj 在第二次调用引用之前已经删除，所以 ref 返回 None。

```
$ python weakref_ref.py
```

```

obj: <__main__.ExpensiveObject object at 0x100da5750>
ref: <weakref at 0x100d99b50; to 'ExpensiveObject' at 0x100da5750>
r(): <__main__.ExpensiveObject object at 0x100da5750>
deleting obj
(Deleting <__main__.ExpensiveObject object at 0x100da5750>)
r(): None

```

2.7.2 引用回调

ref 构造函数接受一个可选的回调函数，删除所引用的对象时会调用这个函数。

```

import weakref

class ExpensiveObject(object):
    def __del__(self):
        print '(Deleting %s)' % self

def callback(reference):
    """Invoked when referenced object is deleted"""
    print 'callback(', reference, ')'

obj = ExpensiveObject()
r = weakref.ref(obj, callback)

print 'obj:', obj
print 'ref:', r
print 'r():', r()

print 'deleting obj'
del obj
print 'r():', r()

```

引用已经“死亡”不再引用原对象时，这个回调会接受引用对象作为参数。这个特性的一种用法就是从缓存删除弱引用对象。

```
$ python weakref_ref_callback.py
```

```
obj: <__main__.ExpensiveObject object at 0x100da1950>
```

```

ref: <weakref at 0x100d99ba8; to 'ExpensiveObject' at 0x100da1950>
r(): <__main__.ExpensiveObject object at 0x100da1950>
deleting obj
callback( <weakref at 0x100d99ba8; dead> )
(Deleting <__main__.ExpensiveObject object at 0x100da1950>)
r(): None

```

2.7.3 代理

有时使用代理比使用弱引用更为方便。使用代理时可以像使用原对象一样，而且不要求访问对象之前先调用代理。这说明，可以将代理传递到一个库，而这个库并不知道它接收的是一个引用而不是真正的对象。

```

import weakref

class ExpensiveObject(object):
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print '(Deleting %s)' % self

obj = ExpensiveObject('My Object')
r = weakref.ref(obj)
p = weakref.proxy(obj)

print 'via obj:', obj.name
print 'via ref:', r().name
print 'via proxy:', p.name
del obj
print 'via proxy:', p.name

```

如果引用对象删除后再访问代理，会产生一个 `ReferenceError` 异常。

```
$ python weakref_proxy.py
```

```

via obj: My Object
via ref: My Object
via proxy: My Object
(Deleting <__main__.ExpensiveObject object at 0x100da27d0>)
via proxy:
Traceback (most recent call last):
  File "weakref_proxy.py", line 26, in <module>
    print 'via proxy:', p.name
ReferenceError: weakly-referenced object no longer exists

```

2.7.4 循环引用

弱引用有一种用法，即在不阻止垃圾回收时允许循环引用。下面的例子展示了图中包含一

个循环时使用常规对象和使用代理的区别。

`weakref_graph.py` 中的 `Graph` 类接受给定的任意对象作为序列中的“下一个”节点。为简洁起见，这个实现支持从各节点有一个“传出”（outgoing）引用，通常这样做用途很有限，不过可以很容易为这些例子创建循环。函数 `demo()` 是一个工具函数，通过创建一个循环然后删除各个引用来尝试使用 `Graph` 类。

```
import gc
from pprint import pprint
import weakref

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.other = None
    def set_next(self, other):
        print '%s.set_next(%r)' % (self.name, other)
        self.other = other
    def all_nodes(self):
        "Generate the nodes in the graph sequence."
        yield self
        n = self.other
        while n and n.name != self.name:
            yield n
            n = n.other
        if n is self:
            yield n
        return
    def __str__(self):
        return '->'.join(n.name for n in self.all_nodes())
    def __repr__(self):
        return '<%s at 0x%x name=%s>' % (self.__class__.__name__,
                                         id(self), self.name)
    def __del__(self):
        print '(Deleting %s)' % self.name
        self.set_next(None)

def collect_and_show_garbage():
    "Show what garbage is present."
    print 'Collecting...'
    n = gc.collect()
    print 'Unreachable objects:', n
    print 'Garbage:',
    pprint(gc.garbage)
def demo(graph_factory):
    print 'Set up graph:'
    one = graph_factory('one')
```



```

two = graph_factory('two')
three = graph_factory('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

print
print 'Graph:'
print str(one)
collect_and_show_garbage()

print
three = None
two = None
print 'After 2 references removed:'
print str(one)
collect_and_show_garbage()

print
print 'Removing last reference:'
one = None
collect_and_show_garbage()

```

这个例子使用 gc 模块帮助调试内存泄露。DEBUG_LEAK 标志会使 gc 打印那些无法看到的对象的有关信息，而不是通过垃圾回收器中这些对象的引用。

```

import gc
from pprint import pprint
import weakref

from weakref_graph import Graph, demo, collect_and_show_garbage

gc.set_debug(gc.DEBUG_LEAK)

print 'Setting up the cycle'
print
demo(Graph)
print
print 'Breaking the cycle and cleaning up garbage'
print
gc.garbage[0].set_next(None)
while gc.garbage:
    del gc.garbage[0]
print
collect_and_show_garbage()

```

即使在 demo() 中删除了 Graph 实例的本地引用，图仍然显示在垃圾列表中，不能回收。而

且垃圾列表中还会找到多个字典。它们是来自 Graph 实例的 `__dict__` 值，包含了这些对象的属性。可以强制删除这些图，因为程序知道它们是什么。向解释器传入 `-u` 选项，启用非缓冲 I/O，从而确保这个示例程序中 `print` 语句的输出（写至标准输出）和 `gc` 的调试输出（写至标准错误输出）正确地隔行显示。

```
$ python -u weakref_cycle.py

Setting up the cycle

Set up graph:
one.set_next(<Graph at 0x100db7590 name=two>)
two.set_next(<Graph at 0x100db75d0 name=three>)
three.set_next(<Graph at 0x100db7550 name=one>)

Graph:
one->two->three->one
Collecting...
Unreachable objects: 0
Garbage: []

After 2 references removed:
one->two->three->one
Collecting...
Unreachable objects: 0
Garbage: []

Removing last reference:
Collecting...
gc: uncollectable <Graph 0x100db7550>
gc: uncollectable <Graph 0x100db7590>
gc: uncollectable <Graph 0x100db75d0>
gc: uncollectable <dict 0x100c63c30>
gc: uncollectable <dict 0x100c5e150>
gc: uncollectable <dict 0x100c63810>
Unreachable objects: 6
Garbage:[<Graph at 0x100db7550 name=one>,
<Graph at 0x100db7590 name=two>,
<Graph at 0x100db75d0 name=three>,
{'name': 'one', 'other': <Graph at 0x100db7590 name=two>},
{'name': 'two', 'other': <Graph at 0x100db75d0 name=three>},
{'name': 'three', 'other': <Graph at 0x100db7550 name=one>}]

Breaking the cycle and cleaning up garbage

one.set_next(None)
(Deleting two)
two.set_next(None)
```

```
(Deleting three)
three.set_next(None)
(Deleting one)
one.set_next(None)

Collecting...
Unreachable objects: 0
Garbage:[]
```

下一步是创建一个更智能的 WeakGraph 类，它知道如何在检测到一个循环时使用弱引用来避免建立常规引用循环。

```
import gc
from pprint import pprint
import weakref
\
from weakref_graph import Graph, demo

class WeakGraph(Graph):
    def set_next(self, other):
        if other is not None:
            # See if we should replace the reference
            # to other with a weakref.
            if self in other.all_nodes():
                other = weakref.proxy(other)
            super(WeakGraph, self).set_next(other)
        return

demo(WeakGraph)
```

由于 WeakGraph 实例使用代理来指示已看到的对象，随着 demo() 删除了对象的所有本地引用，循环会断开，这样垃圾回收器就可以将这些对象删除。

```
$ python weakref_weakgraph.py
```

```
Set up graph:
one.set_next(<WeakGraph at 0x100db4790 name=two>)
two.set_next(<WeakGraph at 0x100db47d0 name=three>)
three.set_next(<weakproxy at 0x100dac6d8 to WeakGraph at 0x100db4750>)
)
```

```
Graph:
one->two->three
Collecting...
Unreachable objects: 0
Garbage:[]
```

```
After 2 references removed:
one->two->three
```

```

Collecting...
Unreachable objects: 0
Garbage: []

Removing last reference:
(Deleting one)
one.set_next(None)
(Deleting two)
two.set_next(None)
(Deleting three)
three.set_next(None)
Collecting...
Unreachable objects: 0
Garbage: []

```

2.7.5 缓存对象

ref 和 proxy 类可以认为是“底层”实现。尽管它们对于维护单个对象的弱引用很有用，并允许将循环垃圾回收，但 WeakKeyDictionary 和 WeakValueDictionary 提供了一个更合适的 API 来创建多个对象的缓存。

WeakValueDictionary 使用其中保存的值的弱引用，当其他代码不再实际使用这些值时允许将其垃圾回收。通过使用垃圾回收器的显式调用，由此说明了使用常规字典和 WeakValueDictionary 完成内存处理的差别。

```

import gc
from pprint import pprint
import weakref

gc.set_debug(gc.DEBUG_LEAK)

class ExpensiveObject(object):
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return 'ExpensiveObject(%s)' % self.name
    def __del__(self):
        print '      (Deleting %s)' % self

def demo(cache_factory):
    # hold objects so any weak references
    # are not removed immediately
    all_refs = {}
    # create the cache using the factory
    print 'CACHE TYPE:', cache_factory
    cache = cache_factory()
    for name in [ 'one', 'two', 'three' ]:

```



```

    o = ExpensiveObject(name)
    cache[name] = o
    all_refs[name] = o
    del o # decref

    print '  all_refs =',
    pprint(all_refs)
    print '\n Before, cache contains:', cache.keys()
    for name, value in cache.items():
        print '    %s = %s' % (name, value)
        del value # decref

    # Remove all references to the objects except the cache
    print '\n Cleanup:'
    del all_refs
    gc.collect()

    print '\n After, cache contains:', cache.keys()
    for name, value in cache.items():
        print '    %s = %s' % (name, value)
    print ' demo returning'
    return

demo(dict)
print

demo(weakref.WeakValueDictionary)

```

如果循环变量指示缓存的值，这些循环变量必须显式清除，从而使对象的引用计数减少。否则，垃圾回收器不会删除这些对象，它们仍会保留在缓存中。类似地，all_refs 变量用来维护引用，避免它们过早地被垃圾回收。

```
$ python weakref_valuedict.py
```

```

CACHE TYPE: <type 'dict'>
  all_refs ={'one': ExpensiveObject(one),
'three': ExpensiveObject(three),
'two': ExpensiveObject(two)}

Before, cache contains: ['three', 'two', 'one']
  three = ExpensiveObject(three)
  two = ExpensiveObject(two)
  one = ExpensiveObject(one)

Cleanup:

After, cache contains: ['three', 'two', 'one']
  three = ExpensiveObject(three)

```



```
two = ExpensiveObject(two)
one = ExpensiveObject(one)
demo returning
    (Deleting ExpensiveObject(three))
    (Deleting ExpensiveObject(two))
    (Deleting ExpensiveObject(one))
CACHE TYPE: weakref.WeakValueDictionary
all_refs = {'one': ExpensiveObject(one),
            'three': ExpensiveObject(three),
            'two': ExpensiveObject(two)}

Before, cache contains: ['three', 'two', 'one']
three = ExpensiveObject(three)
two = ExpensiveObject(two)
one = ExpensiveObject(one)

Cleanup:
    (Deleting ExpensiveObject(three))
    (Deleting ExpensiveObject(two))
    (Deleting ExpensiveObject(one))

After, cache contains: []
demo returning
```

`WeakKeyDictionary` 的工作原理与此类似，不过它使用字典中键的弱引用，而不是字典中的值。

警告： `weakref` 的库文档包含以下警告。

注意： 由于 `WeakValueDictionary` 基于 Python 字典建立，迭代处理它时不能改变大小。对于 `WeakValueDictionary`，可能很难确保这一点，因为程序在迭代期间完成的动作可能会导致字典中的元素“魔法般地”消失（作为垃圾回收的一个副作用）。

参见：

`weakref` (<http://docs.python.org/lib/module-weakref.html>) 这个模块的标准库文档。

`gc` (17.6 节) `gc` 模块是解释器的垃圾回收器的接口。

2.8 copy——复制对象

作用：提供一些函数，可以使用浅副本或深副本语义复制对象。

Python 版本：1.4 及以后版本

`copy` 模块包括两个函数 `copy()` 和 `deepcopy()`，用于复制现有的对象。

2.8.1 浅副本

`copy()` 创建的浅副本（shallow copy）是一个新容器，其中填充原对象内容的引用。建立

list 对象的一个浅副本时，会构造一个新的 list，并将原对象的元素追加到这个 list。

```
import copy

class MyClass:
    def __init__(self, name):
        self.name = name
    def __cmp__(self, other):
        return cmp(self.name, other.name)

a = MyClass('a')
my_list = [ a ]
dup = copy.copy(my_list)

print '          my_list:', my_list
print '          dup:', dup
print '    dup is my_list:', (dup is my_list)
print '    dup == my_list:', (dup == my_list)
print 'dup[0] is my_list[0]:', (dup[0] is my_list[0])
print 'dup[0] == my_list[0]:', (dup[0] == my_list[0])
```

对于一个浅副本，不会复制 MyClass 实例，所以 dup 列表中的引用会指向 my_list 中相同的对象。

```
$ python copy_shallow.py

          my_list: [<__main__.MyClass instance at 0x100dad68>]
          dup: [<__main__.MyClass instance at 0x100dad68>]
    dup is my_list: False
    dup == my_list: True
dup[0] is my_list[0]: True
dup[0] == my_list[0]: True
```

2.8.2 深副本

deepcopy() 创建的深副本是一个新容器，其中填充原对象内容的副本。要建立一个 list 的深副本，会构造一个新的 list，复制原列表的元素，然后将这些副本追加到新列表。

将前例中的 copy() 调用替换为 deepcopy()，可以清楚地看到输出的不同。

```
dup = copy.deepcopy(my_list)
```

列表的第一个元素不再是相同的对象引用，不过比较这两个对象时，仍认为它们是相等的。

```
$ python copy_deep.py

          my_list: [<__main__.MyClass instance at 0x100dad68>]
          dup: [<__main__.MyClass instance at 0x100dad20>]
    dup is my_list: False
    dup == my_list: True
dup[0] is my_list[0]: False
dup[0] == my_list[0]: True
```

2.8.3 定制复制行为

可以使用特殊方法 `__copy__()` 和 `__deepcopy__()` 来控制如何建立副本。

- 调用 `__copy__()` 而不提供任何参数，这会返回对象的一个浅副本。
- 调用 `__deepcopy__()`，并提供一个备忘字典，这会返回对象的一个深副本。所有需要深复制的成员属性都要连同备忘字典传递到 `copy.deepcopy()` 来控制递归。（备忘字典将在后面更详细地解释。）

下面这个例子展示了如何调用这些方法。

```
import copy

class MyClass:
    def __init__(self, name):
        self.name = name
    def __cmp__(self, other):
        return cmp(self.name, other.name)
    def __copy__(self):
        print '__copy__()'
        return MyClass(self.name)
    def __deepcopy__(self, memo):
        print '__deepcopy__(%s)' % str(memo)
        return MyClass(copy.deepcopy(self.name, memo))
```

```
a = MyClass('a')
```

```
sc = copy.copy(a)
```

```
dc = copy.deepcopy(a)
```

备忘字典用于跟踪已复制的值，以避免无限递归。

```
$ python copy_hooks.py
```

```
__copy__()
```

```
__deepcopy__({})
```

2.8.4 深副本中的递归

为了避免复制递归数据结构可能带来的问题，`deepcopy()` 使用了一个字典来跟踪已复制的对象。将这个字典传入 `__deepcopy__()` 方法，从而在该方法中也可以进行检查。

下面的例子显示了一个互连的数据结构（如一个有向图）可以通过实现 `__deepcopy__()` 方法帮助防止递归。

```
import copy
import pprint
```

```
class Graph:
```

```
    def __init__(self, name, connections):
```




```

self.name = name
self.connections = connections

def add_connection(self, other):
    self.connections.append(other)

def __repr__(self):
    return 'Graph(name=%s, id=%s)' % (self.name, id(self))
def __deepcopy__(self, memo):
    print '\nCalling __deepcopy__ for %r' % self
    if self in memo:
        existing = memo.get(self)
        print 'Already copied to %r' % existing
        return existing
    print 'Memo dictionary:'
    pprint.pprint(memo, indent=4, width=40)
    dup = Graph(copy.deepcopy(self.name, memo), [])
    print 'Copying to new object %s' % dup
    memo[self] = dup
    for c in self.connections:
        dup.add_connection(copy.deepcopy(c, memo))
    return dup

root = Graph('root', [])
a = Graph('a', [root])
b = Graph('b', [a, root])
root.add_connection(a)
root.add_connection(b)

dup = copy.deepcopy(root)

```

Graph 类包含一些基本的有向图方法。基于一个名和已连接的现有节点的一个列表可以初始化一个 Graph 实例。add_connection() 方法用于建立双向连接。deepcopy 也用到了这个方法。

__deepcopy__() 方法将打印消息来显示它如何得到调用，并根据需要管理备忘字典内容。它不是复制整个连接列表，而是创建一个新列表，把各个连接的副本追加到这个列表。这样可以确保复制各个新节点时会更新备忘字典，以避免递归问题或多余的节点副本。与前面一样，完成时会返回复制的对象。

图 2.1 中存在几个环，不过利用备忘字典处理递归就可以避免遍历导致栈溢出错误。复制根节点 root 时，输出如下：

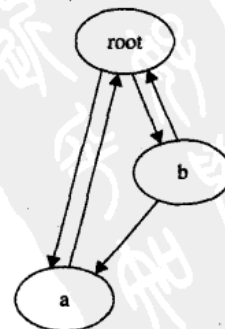


图 2.1 带环对象图的深复制

```

$ python copy_recursion.py

Calling __deepcopy__ for Graph(name=root, id=4309347072)
Memo dictionary:
{ }
Copying to new object Graph(name=root, id=4309347360)

Calling __deepcopy__ for Graph(name=a, id=4309347144)
Memo dictionary:
{ Graph(name=root, id=4309347072): Graph(name=root, id=4309347360),
  4307936896: ['root'],
  4309253504: 'root' }
Copying to new object Graph(name=a, id=4309347504)

Calling __deepcopy__ for Graph(name=root, id=4309347072)
Already copied to Graph(name=root, id=4309347360)

Calling __deepcopy__ for Graph(name=b, id=4309347216)
Memo dictionary:
{ Graph(name=root, id=4309347072): Graph(name=root, id=4309347360),
  Graph(name=a, id=4309347144): Graph(name=a, id=4309347504),
  4307936896: [ 'root',
                'a',
                Graph(name=root, id=4309347072),
                Graph(name=a, id=4309347144)],
  4308678136: 'a',
  4309253504: 'root',
  4309347072: Graph(name=root, id=4309347360),
  4309347144: Graph(name=a, id=4309347504) }
Copying to new object Graph(name=b, id=4309347864)

```

第二次遇到 root 节点时，此时正在复制 a 节点，__deepcopy__() 检测到递归，会重用备忘录中现有的值，而不是创建一个新对象。

参见：

copy (<http://docs.python.org/library/copy.html>) 这个模块的标准库文档。

2.9 pprint——美观打印数据结构

作用：美观打印数据结构。

Python 版本：1.4 及以后版本

pprint 包含一个“美观打印机”（pretty printer），用于生成数据结构的一个美观视图。格式化工具会生成数据结构的一些表示，不仅可以由解释器正确地解析，而且便于人类阅读。输出尽可能放在一行上，分解为多行时则需要缩进。

这一节中的例子都用到了 pprint_data.py，其中包含以下数据。

```
data = [ (1, { 'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D' }),
         (2, { 'e': 'E', 'f': 'F', 'g': 'G', 'h': 'H',
               'i': 'I', 'j': 'J', 'k': 'K', 'l': 'L',
               })],
        ]
```

2.9.1 打印

要使用这个模块，最简单的方法就是利用 `pprint()` 函数。

```
from pprint import pprint

from pprint_data import data

print 'PRINT:'
print data
print
print 'PPRINT:'
pprint(data)
```

`pprint()` 格式化一个对象，并把它写至一个数据流，这个数据流作为参数传入（或者是默认的 `sys.stdout`）。

```
$ python pprint_pprint.py
PRINT:
[(1, {'a': 'A', 'c': 'C', 'b': 'B', 'd': 'D'}), (2, {'e': 'E', 'g':
'G', 'f': 'F', 'i': 'I', 'h': 'H', 'k': 'K', 'j': 'J', 'l': 'L'})]]

PPRINT:
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'})]]
```

2.9.2 格式化

要格式化一个数据结构而不把它直接写至一个流（例如用于日志记录），可以使用 `pformat()` 来构造一个字符串表示。

```
import logging
from pprint import pformat
from pprint_data import data
```

```
logging.basicConfig(level=logging.DEBUG,
                    format='%(levelname)-8s %(message)s',
                    )
```

```
logging.debug('Logging pformatted data')
formatted = pformat(data)
for line in formatted.splitlines():
    logging.debug(line.rstrip())
```

然后可以单独地打印格式化的字符串或者记入日志。

```
$ python pprint_pformat.py
```

```
DEBUG    Logging pformatted data
DEBUG    [(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'})],
DEBUG    (2,
DEBUG    {'e': 'E',
DEBUG    'f': 'F',
DEBUG    'g': 'G',
DEBUG    'h': 'H',
DEBUG    'i': 'I',
DEBUG    'j': 'J',
DEBUG    'k': 'K',
DEBUG    'l': 'L'})]
```

2.9.3 任意类

如果定制类定义了一个 `__repr__()` 方法，`pprint()` 使用的 `PrettyPrinter` 类还可以处理这些定制类。

```
from pprint import pprint

class node(object):
    def __init__(self, name, contents=[]):
        self.name = name
        self.contents = contents[:]
    def __repr__(self):
        return ('node(' + repr(self.name) + ', ' +
                repr(self.contents) + ')')

trees = [ node('node-1',
               [ node('node-2', [ node('node-2-1')]),
                 node('node-3', [ node('node-3-1')])]),
          ]

pprint(trees)
```

由 `PrettyPrinter` 组合嵌套对象的表示，从而返回完整的字符串表示。

```
$ python pprint_arbitrary_object.py
```

```
[node('node-1', []),
 node('node-2', [node('node-2-1', [])]),
 node('node-3', [node('node-3-1', [])])]
```

2.9.4 递归

递归数据结构由指向原数据源的引用来表示，形式为 <Recursion on typename with id=number>。

```
from pprint import pprint

local_data = [ 'a', 'b', 1, 2 ]
local_data.append(local_data)

print 'id(local_data) =>', id(local_data)
pprint(local_data)
```

在这个例子中，列表 `local_data` 增加到其自身，这会创建一个递归引用。

```
$ python pprint_recursion.py

id(local_data) => 4309215280
['a', 'b', 1, 2, <Recursion on list with id=4309215280>]
```

2.9.5 限制嵌套输出

对于非常深的数据结构，可能不要求输出包含所有细节。有可能数据没有适当地格式化，也可能格式化文本过大而无法管理，或者某些数据是多余的。

```
from pprint import pprint

from pprint_data import data

pprint(data, depth=1)
```

使用 `depth` 参数可以控制美观打印机递归处理嵌套数据结构的深度。输出中未包含的层次由一个省略号表示。

```
$ python pprint_depth.py

[(...), (...)]
```

2.9.6 控制输出宽度

格式化文本的默认输出宽度为 80 列。要调整这个宽度，可以在 `pprint()` 中使用参数 `width`。

```
from pprint import pprint
from pprint_data import data

for width in [ 80, 5 ]:
    print 'WIDTH =', width
    pprint(data, width=width)
    print
```

宽度太小不能适应格式化数据结构时，如果截断或转行会引入非法的语法，就不会进行截断或转行。

```
$ python pprint_width.py
```

```
WIDTH = 80
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'}))]

WIDTH = 5
[(1,
  {'a': 'A',
   'b': 'B',
   'c': 'C',
   'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'}))]
```

参见：

`pprint` (<http://docs.python.org/lib/module-pprint.html>)，这个模块的标准库文档。



第 ③ 章

算 法

Python 包含很多模块，可以采用对任务最适用的方式精巧而简洁地实现算法。它支持不同的编程方式，包括纯过程式、面向对象和函数式。这 3 种方式经常在同一个程序的不同部分混合使用。

`functools` 包含的函数用于创建函数修饰符、启用面向方面的编程以及传统面向对象方法所不能支持的代码重用。它还提供了一个类修饰符使用一个快捷方式来实现所有富比较 API，另外提供了 `partial` 对象用来创建函数（包含其参数）的引用。

`itertools` 模块包含的函数用于创建和处理函数式编程中使用的迭代器和生成器。利用 `operator` 模块，通过提供基于函数的内置操作接口，如算术操作或元素查找，使用函数式编程时不再需要很多 `lambda` 函数。

`contextlib` 使得对于所有编程方式来说资源管理会更容易、更可靠，而且更简洁。结合上下文管理器和 `with` 语句，可以减少 `try:finally` 块的个数和所需的缩进层次，同时还能确保文件、套接字、数据库事务和其他资源在适当的时候关闭和释放。

3.1 `functools`——管理函数的工具

作用：处理其他函数的函数。

Python 版本：2.5 及以后版本

`functools` 模块提供了一些工具来调整或扩展函数和其他可回调对象，而不必完全重写。

3.1.1 修饰符

`functools` 模块提供的主要工具是 `partial` 类，它可以用来“包装”一个有默认参数的可回调对象。得到的对象本身是可回调的，可以看作就像是原来的函数。它与原函数的参数完全相同，调用时也可以提供额外的位置或命名参数。可以使用 `partial` 而不是 `lambda` 为函数提供默认参数，有些参数可以不指定。

`partial` 对象

下面这个例子显示了函数 `myfunc()` 的两个简单的 `partial` 对象。`show_details()` 的输出包含这个部分对象的 `func`、`args` 和 `keywords` 属性。

```
import functools
```

```

def myfunc(a, b=2):
    """Docstring for myfunc()."""
    print '  called myfunc with:', (a, b)
    return

def show_details(name, f, is_partial=False):
    """Show details of a callable object."""
    print '%s:' % name
    print '  object:', f
    if not is_partial:
        print '    __name__:', f.__name__
    if is_partial:
        print '    func:', f.func
        print '    args:', f.args
        print '    keywords:', f.keywords
    return

show_details('myfunc', myfunc)
myfunc('a', 3)
print

# Set a different default value for 'b', but require
# the caller to provide 'a'.
p1 = functools.partial(myfunc, b=4)
show_details('partial with named default', p1, True)
p1('passing a')
p1('override b', b=5)
print

# Set default values for both 'a' and 'b'.
p2 = functools.partial(myfunc, 'default a', b=99)
show_details('partial with defaults', p2, True)
p2()
p2(b='override b')
print

print 'Insufficient arguments:'
p1()

```

在这个例子的最后，调用了之前创建的第一个 `partial`，但没有为 `a` 传入一个值，这就导致一个异常。

```
$ python functools_partial.py
```

```

myfunc:
  object: <function myfunc at 0x100d9bf50>
  __name__: myfunc
  called myfunc with: ('a', 3)

```



```

partial with named default:
  object: <functools.partial object at 0x100d993c0>
  func: <function myfunc at 0x100d9bf50>
  args: ()
  keywords: {'b': 4}
  called myfunc with: ('passing a', 4)
  called myfunc with: ('override b', 5)

```

```

partial with defaults:
  object: <functools.partial object at 0x100d99418>
  func: <function myfunc at 0x100d9bf50>
  args: ('default a',)
  keywords: {'b': 99}
  called myfunc with: ('default a', 99)
  called myfunc with: ('default a', 'override b')

```

```

Insufficient arguments:
Traceback (most recent call last):
  File "functools_partial.py", line 51, in <module>
    p1()
TypeError: myfunc() takes at least 1 argument (1 given)

```

获取函数属性

默认情况下，partial 对象没有 `__name__` 或 `__doc__` 属性。如果没有这些属性，修饰的函数将更难调试。使用 `update_wrapper()` 可以从原函数将属性复制或添加到 partial 对象。

```

import functools

def myfunc(a, b=2):
    """Docstring for myfunc()."""
    print ' called myfunc with:', (a, b)
    return

def show_details(name, f):
    """Show details of a callable object."""
    print '%s:' % name
    print ' object:', f
    print ' __name__:',
    try:
        print f.__name__
    except AttributeError:
        print '(no __name__)'
    print ' __doc__', repr(f.__doc__)
    print
    return

```



```

show_details('myfunc', myfunc)

p1 = functools.partial(myfunc, b=4)
show_details('raw wrapper', p1)

print 'Updating wrapper:'
print '  assign:', functools.WRAPPER_ASSIGNMENTS
print '  update:', functools.WRAPPER_UPDATES
print

functools.update_wrapper(p1, myfunc)
show_details('updated wrapper', p1)

```

添加到包装器的属性在 WRAPPER_ASSIGNMENTS 中定义，而 WRAPPER_UPDATES 列出了要修改的值。

```

$ python functools_update_wrapper.py

myfunc:
  object: <function myfunc at 0x100da2050>
  __name__: myfunc
  __doc__ 'Docstring for myfunc().'

raw wrapper:
  object: <functools.partial object at 0x100d993c0>
  __name__: (no __name__)
  __doc__ 'partial(func, *args, **keywords) - new function with parti
al application\n    of the given arguments and keywords.\n'

Updating wrapper:
  assign: ('__module__', '__name__', '__doc__')
  update: ('__dict__',)

updated wrapper:
  object: <functools.partial object at 0x100d993c0>
  __name__: myfunc
  __doc__ 'Docstring for myfunc().'

```

其他可回调对象

Partial 适用于任何可回调对象，而不只是单独的函数。

```

import functools

class MyClass(object):
    """Demonstration class for functools"""

    def method1(self, a, b=2):
        """Docstring for method1()."""

```

```
        print ' called method1 with:', (self, a, b)
        return

    def method2(self, c, d=5):
        """Docstring for method2"""
        print ' called method2 with:', (self, c, d)
        return
    wrapped_method2 = functools.partial(method2, 'wrapped c')
    functools.update_wrapper(wrapped_method2, method2)

    def __call__(self, e, f=6):
        """Docstring for MyClass.__call__"""
        print ' called object with:', (self, e, f)
        return

def show_details(name, f):
    """Show details of a callable object."""
    print '%s:' % name
    print ' object:', f
    print ' __name__:',
    try:
        print f.__name__
    except AttributeError:
        print '(no __name__)'
    print ' __doc__', repr(f.__doc__)
    return

o = MyClass()

show_details('method1 straight', o.method1)
o.method1('no default for a', b=3)
print

p1 = functools.partial(o.method1, b=4)
functools.update_wrapper(p1, o.method1)
show_details('method1 wrapper', p1)
p1('a goes here')
print

show_details('method2', o.method2)
o.method2('no default for c', d=6)
print

show_details('wrapped method2', o.wrapped_method2)
o.wrapped_method2('no default for c', d=6)
print
```



```
show_details('instance', o)
o('no default for e')
print
p2 = functools.partial(o, f=7)
show_details('instance wrapper', p2)
p2('e goes here')
```

这个例子由一个实例以及实例的方法来创建部分对象。

```
$ python functools_method.py
```

method1 straight:

```
object: <bound method MyClass.method1 of <__main__.MyClass object
at 0x100da3550>>
__name__: method1
__doc__ 'Docstring for method1().'
called method1 with: (<__main__.MyClass object at 0x100da3550>, 'n
o default for a', 3)
```

method1 wrapper:

```
object: <functools.partial object at 0x100d99470>
__name__: method1
__doc__ 'Docstring for method1().'
called method1 with: (<__main__.MyClass object at 0x100da3550>, 'a
goes here', 4)
```

method2:

```
object: <bound method MyClass.method2 of <__main__.MyClass object
at 0x100da3550>>
__name__: method2
__doc__ 'Docstring for method2'
called method2 with: (<__main__.MyClass object at 0x100da3550>, 'n
o default for c', 6)
```

wrapped method2:

```
object: <functools.partial object at 0x100d993c0>
__name__: method2
__doc__ 'Docstring for method2'
called method2 with: ('wrapped c', 'no default for c', 6)
```

instance:

```
object: <__main__.MyClass object at 0x100da3550>
__name__: (no __name__)
__doc__ 'Demonstration class for functools'
called object with: (<__main__.MyClass object at 0x100da3550>, 'no
default for e', 6)
```

instance wrapper:

```
object: <functools.partial object at 0x100d994c8>
```

```

__name__: (no __name__)
__doc__ 'partial(func, *args, **keywords) - new function with part
ial application\n      of the given arguments and keywords.\n'
called object with: (<__main__.MyClass object at 0x100da3550>, 'e
goes here', 7)

```

为修饰符获取函数属性

在修饰符中使用时，更新包装的可回调对象的属性尤其有用，因为变换后的函数最后会得到原“裸”函数的属性。

```
import functools
```

```

def show_details(name, f):
    """Show details of a callable object."""
    print '%s:' % name
    print '  object:', f
    print '  __name__:',
    try:
        print f.__name__
    except AttributeError:
        print '(no __name__)'
    print '  __doc__', repr(f.__doc__)
    print
    return

def simple_decorator(f):
    @functools.wraps(f)
    def decorated(a='decorated defaults', b=1):
        print '  decorated:', (a, b)
        print '  ',
        f(a, b=b)
        return
    return decorated

def myfunc(a, b=2):
    "myfunc() is not complicated"
    print '  myfunc:', (a,b)
    return
# The raw function
show_details('myfunc', myfunc)
myfunc('unwrapped, default b')
myfunc('unwrapped, passing b', 3)
print

# Wrap explicitly
wrapped_myfunc = simple_decorator(myfunc)
show_details('wrapped_myfunc', wrapped_myfunc)

```



```
wrapped_myfunc()
wrapped_myfunc('args to wrapped', 4)
print

# Wrap with decorator syntax
@simple_decorator
def decorated_myfunc(a, b):
    myfunc(a, b)
    return

show_details('decorated_myfunc', decorated_myfunc)
decorated_myfunc()
decorated_myfunc('args to decorated', 4)
```

functools 提供了一个修饰符 `wraps()`，它会对所修饰的函数应用 `update_wrapper()`。

```
$ python functools_wraps.py
```

```
myfunc:
  object: <function myfunc at 0x100da3488>
  __name__: myfunc
  __doc__ 'myfunc() is not complicated'

  myfunc: ('unwrapped, default b', 2)
  myfunc: ('unwrapped, passing b', 3)

wrapped_myfunc:
  object: <function myfunc at 0x100da3500>
  __name__: myfunc
  __doc__ 'myfunc() is not complicated'

  decorated: ('decorated defaults', 1)
    myfunc: ('decorated defaults', 1)
  decorated: ('args to wrapped', 4)
    myfunc: ('args to wrapped', 4)

decorated_myfunc:
  object: <function decorated_myfunc at 0x100da35f0>
  __name__: decorated_myfunc
  __doc__ None

  decorated: ('decorated defaults', 1)
    myfunc: ('decorated defaults', 1)
  decorated: ('args to decorated', 4)
    myfunc: ('args to decorated', 4)
```



3.1.2 比较

在 Python 2 中，类可以定义一个 `__cmp__()` 方法，后者会根据这个对象小于、等于还是大于所比较的元素返回 -1、0 或 1。Python 2.1 引入了富比较（rich comparison）方法 API（`__lt__()`、`__le__()`、`__eq__()`、`__ne__()`、`__gt__()` 和 `__ge__()`），可以完成一个比较操作并返回一个 Boolean 值。Python 3 则废弃了 `__cmp__()` 而代之以这些新方法，所以 `functools` 提供了一些工具，以便于编写 Python 2 类而且同时符合 Python 3 中新的比较需求。

富比较

设计富比较 API 是为了支持涉及复杂比较的类，从而以最高效的方式实现各个测试。不过，对于比较相对简单的类，手动地创建各个富比较方法就没有必要了。`total_ordering()` 类修饰符取一个提供了部分方法的类，并添加其余的方法。

```
import functools
import inspect
from pprint import pprint

@functools.total_ordering
class MyObject(object):
    def __init__(self, val):
        self.val = val
    def __eq__(self, other):
        print 'testing __eq__ (%s, %s)' % (self.val, other.val)
        return self.val == other.val
    def __gt__(self, other):
        print 'testing __gt__ (%s, %s)' % (self.val, other.val)
        return self.val > other.val

print 'Methods:\n'
pprint(inspect.getmembers(MyObject, inspect.ismethod))

a = MyObject(1)
b = MyObject(2)

print '\nComparisons:'
for expr in [ 'a < b', 'a <= b', 'a == b', 'a >= b', 'a > b' ]:
    print '\n%-6s:' % expr
    result = eval(expr)
    print 'result of %s: %s' % (expr, result)
```

这个类必须提供 `__eq__()` 和另外一个富比较方法的实现。修饰符会增加其余方法的实现，它们要利用类提供的比较来完成工作。

```
$ python functools_total_ordering.py
```

```
Methods:
```

```
[('__eq__', <unbound method MyObject.__eq__>),
 ('__ge__', <unbound method MyObject.__ge__>),
 ('__gt__', <unbound method MyObject.__gt__>),
 ('__init__', <unbound method MyObject.__init__>),
 ('__le__', <unbound method MyObject.__le__>),
 ('__lt__', <unbound method MyObject.__lt__>)]
```

Comparisons:

```
a < b :
    testing __gt__(2, 1)
    result of a < b: True

a <= b:
    testing __gt__(1, 2)
    result of a <= b: True

a == b:
    testing __eq__(1, 2)
    result of a == b: False
a >= b:
    testing __gt__(2, 1)
    result of a >= b: False

a > b :
    testing __gt__(1, 2)
    result of a > b: False
```

比对序

由于 Python 3 中废弃了老式比较函数，因此 `sort()` 之类的函数中也不再支持 `cmp` 参数。对于使用了比较函数的 Python 2 程序，可以用 `cmp_to_key()` 将它们转换为一个返回比对键 (collation key) 的函数，这个键用于确定元素在最终序列中的位置。

```
import functools

class MyObject(object):
    def __init__(self, val):
        self.val = val
    def __str__(self):
        return 'MyObject(%s)' % self.val

def compare_obj(a, b):
    """Old-style comparison function.
    """
    print 'comparing %s and %s' % (a, b)
    return cmp(a.val, b.val)
```




```
# Make a key function using cmp_to_key()
get_key = functools.cmp_to_key(compare_obj)

def get_key_wrapper(o):
    """Wrapper function for get_key to allow for print statements.
    """
    new_key = get_key(o)
    print 'key_wrapper(%s) -> %s' % (o, new_key)
    return new_key
objs = [ MyObject(x) for x in xrange(5, 0, -1) ]

for o in sorted(objs, key=get_key_wrapper):
    print o
```

正常情况下，可以直接使用 `cmp_to_key()`，不过这个例子中引入了一个额外的包装器函数，从而在调用 `key` 函数时可以输出更多的信息。

如输出所示，`sorted()` 首先对序列中的每一个元素调用 `get_key_wrapper()` 来生成一个键。`cmp_to_key()` 返回的键是 `functools` 中定义的一个类的实例，这个类使用所传入的老式比较函数来实现富比较 API。所有键都创建之后，将通过比较这些键对序列排序。

```
$ python functools_cmp_to_key.py
```

```
key_wrapper(MyObject(5)) -> <functools.K object at 0x100da2a50>
key_wrapper(MyObject(4)) -> <functools.K object at 0x100da2a90>
key_wrapper(MyObject(3)) -> <functools.K object at 0x100da2ad0>
key_wrapper(MyObject(2)) -> <functools.K object at 0x100da2b10>
key_wrapper(MyObject(1)) -> <functools.K object at 0x100da2b50>
comparing MyObject(4) and MyObject(5)
comparing MyObject(3) and MyObject(4)
comparing MyObject(2) and MyObject(3)
comparing MyObject(1) and MyObject(2)
MyObject(1)
MyObject(2)
MyObject(3)
MyObject(4)
MyObject(5)
```

参见：

`functools` (<http://docs.python.org/library/functools.html>) 这个模块的标准库文档。

Rich comparison methods (http://docs.python.org/reference/datamodel.html#object.__lt__)

Python 参考指南中对富比较方法的描述。

`inspect` (18.4 节) 活动对象的自省 API。

3.2 itertools——迭代器函数

作用: itertools 模块包含一组函数用于处理序列数据集。

Python 版本: 2.3 及以后版本

itertools 提供的函数是受函数式编程语言（如 Clojure 和 Haskell）中类似特性的启发。其目的是保证快速，并且高效地使用内存，而且可以联结在一起表述更为复杂的基于迭代的算法。

与使用列表的代码相比，基于迭代器的算法可以提供更好的内存使用特性。在真正需要数据之前，并不从迭代器生成数据，由于这个原因，不需要将所有数据都同时存储在内存中。这种“懒”处理模型可以减少内存使用，相应地还可以减少交换以及大数据集的其他副作用，从而改善性能。

3.2.1 合并和分解迭代器

chain() 函数取多个迭代器作为参数，最后返回一个迭代器，它能生成所有输入迭代器的内容，就好像这些内容来自一个迭代器一样。

```
from itertools import *

for i in chain([1, 2, 3], ['a', 'b', 'c']):
    print i,
print
```

利用 chain()，可以轻松的处理多个序列而不必构造一个大的列表。

```
$ python itertools_chain.py
```

```
1 2 3 a b c
```

izip() 返回一个迭代器，它会把多个迭代器的元素结合到一个元组中。

```
from itertools import *

for i in izip([1, 2, 3], ['a', 'b', 'c']):
    print i
```

这个函数的工作方式类似于内置函数 zip()，只不过它会返回一个迭代器而不是一个列表。

```
$ python itertools_izip.py
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

islice() 函数返回一个迭代器，它按索引返回由输入迭代器所选的元素。

```
from itertools import *

print 'Stop at 5:'
for i in islice(count(), 5):
```

```

    print i,
    print '\n'

    print 'Start at 5, Stop at 10:'
    for i in islice(count(), 5, 10):
        print i,
    print '\n'

    print 'By tens to 100:'
    for i in islice(count(), 0, 100, 10):
        print i,
    print '\n'

```

islice() 与列表的 slice 操作符参数相同，同样包括开始位置 (start)、结束位置 (stop) 和步长 (step)。start 和 step 参数是可选的。

```
$ python itertools_islice.py
```

```

Stop at 5:
0 1 2 3 4

```

```

Start at 5, Stop at 10:
5 6 7 8 9

```

```

By tens to 100:
0 10 20 30 40 50 60 70 80 90

```

tee() 函数根据一个原输入迭代器返回多个独立的迭代器 (默认为两个)。

```

from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)

print 'i1:', list(i1)
print 'i2:', list(i2)

```

tee() 的语义类似于 UNIX tee 工具，它会重复从输入读到的值，并把它们写至一个命名文件和标准输出。tee() 返回的迭代器可以用来为将并行处理的多个算法提供相同的数据集。

```
$ python itertools_tee.py
```

```

i1: [0, 1, 2, 3, 4]
i2: [0, 1, 2, 3, 4]

```

tee() 创建的新迭代器共享其输入迭代器，所以一旦创建了新迭代器，就不应再使用原迭代器。

```

from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)

```

```

print 'r:',
for i in r:
    print i,
    if i > 1:
        break
print

print 'i1:', list(i1)
print 'i2:', list(i2)

```

如果原输入迭代器的值已经利用，新迭代器就不会再生成这些值。

```
$ python itertools_tee_error.py
```

```

r: 0 1 2
i1: [3, 4]
i2: [3, 4]

```

3.2.2 转换输入

`imap()` 函数会返回一个迭代器，它对输入迭代器中的值调用一个函数并返回结果。`imap()` 函数的工作方式类似于内置函数 `map()`，只不过只要有某个输入迭代器中的元素全部用完，`imap()` 函数都会停止（而不是插入 `None` 值来完全利用所有输入）。

```

from itertools import *

print 'Doubles:'
for i in imap(lambda x:2*x, xrange(5)):
    print i

print 'Multiples:'
for i in imap(lambda x,y:(x, y, x*y), xrange(5), xrange(5,10)):
    print '%d * %d = %d' % i

```

在第一个例子中，`lambda` 函数将输入值乘以 2。在第二个例子中，`lambda` 函数将两个参数相乘（这两个参数分别来自不同的迭代器），并返回一个 `tuple`，其中包含原参数和计算得到的值。

```
$ python itertools_imap.py
```

```

Doubles:
0
2
4
6
8
Multiples:
0 * 5 = 0
1 * 6 = 6

```

```
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

starmap() 函数类似于 imap(), 不过并不是由多个迭代器构建一个 tuple, 它使用 * 语法分解一个迭代器中的元素作为映射函数的参数。

```
from itertools import *

values = [(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
for i in starmap(lambda x,y:(x, y, x*y), values):
    print '%d * %d = %d' % i
```

imap() 的映射函数名为 f(i1, i2), 而传入 starmap() 的映射函数名为 f(*i)。

```
$ python itertools_starmap.py
```

```
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

3.2.3 生成新值

count() 函数返回一个迭代器, 能够无限地生成连续整数。第一个数可以作为参数传入 (默认为 0)。这里没有上界参数 [参见内置 xrange() 来更多地控制结果集]。

```
from itertools import *

for i in izip(count(1), ['a', 'b', 'c']):
    print i
```

这个例子会停止, 因为 list 参数已经利用过。

```
$ python itertools_count.py
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

cycle() 函数返回一个迭代器, 它会无限地重复给定参数的内容。由于必须记住输入迭代器的全部内容, 因此如果这个迭代器很长, 可能会耗费大量内存。

```
from itertools import *

for i, item in izip(xrange(7), cycle(['a', 'b', 'c'])):
    print (i, item)
```

这个例子中使用了一个计数器变量, 在数个周期后会跳出循环。

```
$ python itertools_cycle.py
```

```
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'a')
(4, 'b')
(5, 'c')
(6, 'a')
```

repeat() 函数返回一个迭代器，每次访问时会生成相同的值。

```
from itertools import *
```

```
for i in repeat('over-and-over', 5):
    print i
```

repeat() 返回的迭代器会一直返回数据，除非提供了可选的 times 参数来限制次数。

```
$ python itertools_repeat.py
```

```
over-and-over
over-and-over
over-and-over
over-and-over
over-and-over
```

如果既要包含来自其他迭代器的值，还要包含一些不变的值，那么将 repeat() 与 izip() 或 imap() 结合使用会很有用。

```
from itertools import *
```

```
for i, s in izip(count(), repeat('over-and-over', 5)):
    print i, s
```

这个例子中就结合了一个计数器值和 repeat() 返回的常量。

```
$ python itertools_repeat_izip.py
```

```
0 over-and-over
1 over-and-over
2 over-and-over
3 over-and-over
4 over-and-over
```

下面这个例子使用 imap() 将 0 ~ 4 区间中的数乘以 2。

```
from itertools import *
```

```
for i in imap(lambda x,y:(x, y, x*y), repeat(2), xrange(5)):
    print '%d * %d = %d' % i
```

repeat() 迭代器不需要显式限制，因为任何一个输入结束时 imap() 就会停止处理，而 xrange() 只返回 5 个元素。

```
$ python itertools_repeat_imap.py

2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

3.2.4 过滤

`dropwhile()` 函数返回一个迭代器，它会生成输入迭代器中条件第一次为 `false` 之后的元素。

```
from itertools import *

def should_drop(x):
    print 'Testing:', x
    return (x<1)

for i in dropwhile(should_drop, [ -1, 0, 1, 2, -2 ]):
    print 'Yielding:', i
```

`dropwhile()` 并不会过滤输入的每一个元素；第一次条件为 `false` 之后，输入中所有余下的元素都会返回。

```
$ python itertools_dropwhile.py

Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Yielding: 2
Yielding: -2
```

`takewhile()` 与 `dropwhile()` 正好相反。它也返回一个迭代器，这个迭代器将返回输入迭代器中保证测试条件为 `true` 的元素。

```
from itertools import *

def should_take(x):
    print 'Testing:', x
    return (x<2)

for i in takewhile(should_take, [ -1, 0, 1, 2, -2 ]):
    print 'Yielding:', i
```

一旦 `should_take()` 返回 `false`, `takewhile()` 就会停止处理输入。

```
$ python itertools_takewhile.py

Testing: -1
Yielding: -1
```

```
Testing: 0
Yielding: 0
Testing: 1
Yielding: 1
Testing: 2
```

`ifilter()` 返回一个迭代器，它的工作方式与内置 `filter()` 处理列表的做法类似，其中只包含测试条件返回 `true` 时的相应元素。

```
from itertools import *

def check_item(x):
    print 'Testing:', x
    return (x<1)

for i in ifilter(check_item, [ -1, 0, 1, 2, -2 ]):
    print 'Yielding:', i
```

`ifilter()` 与 `dropwhile()` 不同，在返回之前对每一个元素都会进行测试。

```
$ python itertools_ifilter.py
```

```
Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Testing: 2
Testing: -2
Yielding: -2
```

`ifilterfalse()` 返回一个迭代器，其中只包含测试条件返回 `false` 时的相应元素。

```
from itertools import *

def check_item(x):
    print 'Testing:', x
    return (x<1)

for i in ifilterfalse(check_item, [ -1, 0, 1, 2, -2 ]):
    print 'Yielding:', i
```

`check_item()` 中的测试表达式与前面相同，所以在这个使用 `ifilterfalse()` 的例子中，其结果与上一个例子的结果正好相反。

```
$ python itertools_ifilterfalse.py
```

```
Testing: -1
Testing: 0
Testing: 1
Yielding: 1
```



```

Testing: 2
Yielding: 2
Testing: -2

```

3.2.5 数据分组

groupby() 函数返回一个迭代器，它会生成按一个公共键组织的值集。下面这个例子展示了如何根据一个属性对相关的值分组。

```

from itertools import *
import operator
import pprint

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return '(%s, %s)' % (self.x, self.y)
    def __cmp__(self, other):
        return cmp((self.x, self.y), (other.x, other.y))

# Create a dataset of Point instances
data = list(imap(Point,
                  cycle(islice(count(), 3)),
                  islice(count(), 7),
                  ))

print 'Data:'
pprint.pprint(data, width=69)
print

# Try to group the unsorted data based on X values
print 'Grouped, unsorted:'
for k, g in groupby(data, operator.attrgetter('x')):
    print k, list(g)
print

# Sort the data
data.sort()
print 'Sorted:'
pprint.pprint(data, width=69)
print

# Group the sorted data based on X values
print 'Grouped, sorted:'
for k, g in groupby(data, operator.attrgetter('x')):
    print k, list(g)
print

```

输入序列要根据键值排序，以保证得到预期的分组。

```
$ python itertools_groupby_seq.py
```

Data:

```
[(0, 0),
 (1, 1),
 (2, 2),
 (0, 3),
 (1, 4),
 (2, 5),
 (0, 6),
 (1, 7),
 (2, 8),
 (0, 9)]
```

Grouped, unsorted:

```
0 [(0, 0)]
1 [(1, 1)]
2 [(2, 2)]
0 [(0, 3)]
1 [(1, 4)]
2 [(2, 5)]
0 [(0, 6)]
1 [(1, 7)]
2 [(2, 8)]
0 [(0, 9)]
```

Sorted:

```
[(0, 0),
 (0, 3),
 (0, 6),
 (0, 9),
 (1, 1),
 (1, 4),
 (1, 7),
 (2, 2),
 (2, 5),
 (2, 8)]
```

Grouped, sorted:

```
0 [(0, 0), (0, 3), (0, 6), (0, 9)]
1 [(1, 1), (1, 4), (1, 7)]
2 [(2, 2), (2, 5), (2, 8)]
```

参见:

itertools (<http://docs.python.org/library/itertools.html>) 这个模块的标准库文档。



The Standard ML Basis Library (www.standardml.org/Basis/) SML 库。

Definition of Haskell and the Standard Libraries (www.haskell.org/definition/) 函数语言 Haskell 的标准库规范。

Clojure (<http://clojure.org/>) Clojure 是一种在 Java 虚拟机上运行的动态函数语言。

tee (<http://unixhelp.ed.ac.uk/CGI/man-cgi?tee>) 这是一个 UNIX 命令行工具，用于将一个输入分解为多个相同的输出流。

3.3 operator——内置操作符的函数接口

作用：内置操作符的函数接口。

Python 版本：1.4 及以后版本

使用迭代器编程时，有时需要为简单的表达式创建小函数。有些情况下，这些确实可以实现为 lambda 函数，不过对于某些操作根本不需要新函数。operator 模块定义了一些对应算术和比较内置操作的函数。

3.3.1 逻辑操作

有一些函数可以用来确定一个值的相应 Boolean 值，将其取反来创建相反的 Boolean 值，以及比较对象查看它们是否相等。

```
from operator import *

a = -1
b = 5

print 'a =', a
print 'b =', b
print

print 'not_(a)      : ', not_(a)
print 'truth(a)     : ', truth(a)
print 'is_(a, b)    : ', is_(a,b)
print 'is_not(a, b): ', is_not(a,b)
```

not_() 后面有下划线，因为 not 是一个 Python 关键字。truth() 会应用 if 语句中测试一个表达式时所用的同样的逻辑。is_() 实现了 is 关键字使用的相同检查，is_not() 完成同样的测试，但返回相反的答案。

```
$ python operator_boolean.py

a = -1
b = 5

not_(a)      : False
```

```
truth(a)      : True
is_(a, b)     : False
is_not(a, b)  : True
```

3.3.2 比较操作符

所有富比较操作符都得到支持。

```
from operator import *
```

```
a = 1
b = 5.0
print 'a =', a
print 'b =', b
for func in (lt, le, eq, ne, ge, gt):
    print '%s(a, b):' % func.__name__, func(a, b)
```

这些函数等价于使用 <、<=、==、>= 和 > 的表达式语法。

```
$ python operator_comparisons.py
```

```
a = 1
b = 5.0
lt(a, b): True
le(a, b): True
eq(a, b): False
ne(a, b): True
ge(a, b): False
gt(a, b): False
```

3.3.3 算术操作符

处理数字值的算术操作符也得到支持。

```
from operator import *
```

```
a = -1
b = 5.0
c = 2
d = 6

print 'a =', a
print 'b =', b
print 'c =', c
print 'd =', d

print '\nPositive/Negative:'
print 'abs(a):', abs(a)
print 'neg(a):', neg(a)
print 'neg(b):', neg(b)
```



```

print 'pos(a):', pos(a)
print 'pos(b):', pos(b)
print '\nArithmetic:'
print 'add(a, b)      :', add(a, b)
print 'div(a, b)      :', div(a, b)
print 'div(d, c)      :', div(d, c)
print 'floordiv(a, b):', floordiv(a, b)
print 'floordiv(d, c):', floordiv(d, c)
print 'mod(a, b)      :', mod(a, b)
print 'mul(a, b)      :', mul(a, b)
print 'pow(c, d)      :', pow(c, d)
print 'sub(b, a)      :', sub(b, a)
print 'truediv(a, b) :', truediv(a, b)
print 'truediv(d, c) :', truediv(d, c)

print '\nBitwise:'
print 'and_(c, d)     :', and_(c, d)
print 'invert(c)      :', invert(c)
print 'lshift(c, d):', lshift(c, d)
print 'or_(c, d)      :', or_(c, d)
print 'rshift(d, c):', rshift(d, c)
print 'xor(c, d)      :', xor(c, d)

```

有两个不同的除法操作符：floordiv()（Python 3.0 版本之前实现的整数除法）和 truediv()（浮点数除法）。

```
$ python operator_math.py
```

```

a = -1
b = 5.0
c = 2
d = 6

```

Positive/Negative:

```

abs(a): 1
neg(a): 1
neg(b): -5.0
pos(a): -1
pos(b): 5.0

```

Arithmetic:

```

add(a, b)      : 4.0
div(a, b)      : -0.2
div(d, c)      : 3
floordiv(a, b): -1.0
floordiv(d, c): 3
mod(a, b)      : 4.0

```



```

mul(a, b)      : -5.0
pow(c, d)      : 64
sub(b, a)      : 6.0
truediv(a, b)  : -0.2
truediv(d, c)  : 3.0

```

```

Bitwise:
and_(c, d)     : 2
invert(c)      : -3
lshift(c, d)   : 128
or_(c, d)      : 6
rshift(d, c)   : 1
xor(c, d)      : 4

```

3.3.4 序列操作符

处理序列的操作符可以划分为 4 组：建立序列、搜索元素、访问内容和从序列删除元素。

```
from operator import *
```

```

a = [ 1, 2, 3 ]
b = [ 'a', 'b', 'c' ]

```

```

print 'a =', a
print 'b =', b

```

```

print '\nConstructive:'
print ' concat(a, b):', concat(a, b)
print ' repeat(a, 3):', repeat(a, 3)

```

```

print '\nSearching:'
print ' contains(a, 1) :', contains(a, 1)
print ' contains(b, "d"):', contains(b, "d")
print ' countOf(a, 1) :', countOf(a, 1)
print ' countOf(b, "d") :', countOf(b, "d")
print ' indexOf(a, 5) :', indexOf(a, 1)

```

```

print '\nAccess Items:'
print ' getitem(b, 1) :', getitem(b, 1)
print ' getslice(a, 1, 3) :', getslice(a, 1, 3)
print ' setitem(b, 1, "d") :', setitem(b, 1, "d"),
print ', after b =', b
print ' setslice(a, 1, 3, [4, 5]):', setslice(a, 1, 3, [4, 5]),
print ', after a =', a

```

```

print '\nDestructive:'
print ' delitem(b, 1) :', delitem(b, 1), ', after b =', b
print ' delslice(a, 1, 3):', delslice(a, 1, 3), ', after a =', a

```

其中一些操作（如 `setitem()` 和 `delitem()`）会原地修改序列，而不返回任何值。

```
$ python operator_sequences.py
```

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
```

Constructive:

```
concat(a, b): [1, 2, 3, 'a', 'b', 'c']
repeat(a, 3): [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Searching:

```
contains(a, 1) : True
contains(b, "d"): False
countOf(a, 1)  : 1
countOf(b, "d") : 0
indexOf(a, 5)   : 0
```

Access Items:

```
getitem(b, 1)           : b
getslice(a, 1, 3)       : [2, 3]
setitem(b, 1, "d")      : None , after b = ['a', 'd', 'c']
setslice(a, 1, 3, [4, 5]): None , after a = [1, 4, 5]
```

Destructive:

```
delitem(b, 1)          : None , after b = ['a', 'c']
delslice(a, 1, 3): None , after a = [1]
```

3.3.5 原地操作符

除了标准操作符外，很多对象类型还通过一些特殊操作符（如 `+=`）支持“原地”修改。这些原地修改也有相应的等价函数。

```
from operator import *
```

```
a = -1
b = 5.0
c = [ 1, 2, 3 ]
d = [ 'a', 'b', 'c' ]
print 'a =', a
print 'b =', b
print 'c =', c
print 'd =', d
print

a = iadd(a, b)
print 'a = iadd(a, b) =>', a
print
```



```
c = iconcat(c, d)
print 'c = iconcat(c, d) =>', c
```

这些例子只展示了部分函数。要全面了解有关详细内容，请参考标准库文档。

```
$ python operator_inplace.py
```

```
a = -1
b = 5.0
c = [1, 2, 3]
d = ['a', 'b', 'c']

a = iadd(a, b) => 4.0

c = iconcat(c, d) => [1, 2, 3, 'a', 'b', 'c']
```

3.3.6 属性和元素“获取方法”

`operator` 模块最特别的特性之一是获取方法 (getter) 的概念。获取方法是运行时构造的一些可回调对象，用来获取对象的属性或序列的内容。获取方法在处理迭代器或生成器序列时特别有用，它们引入的开销会大大低于 `lambda` 或 Python 函数的开销。

```
from operator import *

class MyObj(object):
    """example class for attrgetter"""
    def __init__(self, arg):
        super(MyObj, self).__init__()
        self.arg = arg
    def __repr__(self):
        return 'MyObj(%s)' % self.arg

l = [ MyObj(i) for i in xrange(5) ]
print 'objects : ', l

# Extract the 'arg' value from each object
g = attrgetter('arg')
vals = [ g(i) for i in l ]
print 'arg values:', vals

# Sort using arg
l.reverse()
print 'reversed : ', l
print 'sorted : ', sorted(l, key=g)
```

属性获取方法的工作类似于 `lambda x, n='attrname': getattr(x, n)`:

```
$ python operator_attrgetter.py
```




```
objects : [MyObj(0), MyObj(1), MyObj(2), MyObj(3), MyObj(4)]
arg values: [0, 1, 2, 3, 4]
reversed : [MyObj(4), MyObj(3), MyObj(2), MyObj(1), MyObj(0)]
sorted : [MyObj(0), MyObj(1), MyObj(2), MyObj(3), MyObj(4)]
```

元素获取方法的工作类似于 `lambda x, y=5: x[y]`:

```
from operator import *

l = [ dict(val=-1 * i) for i in xrange(4) ]
print 'Dictionaries:', l
g = itemgetter('val')
vals = [ g(i) for i in l ]
print '      values:', vals
print '      sorted:', sorted(l, key=g)
print
l = [ (i, i*-2) for i in xrange(4) ]
print 'Tuples      :', l
g = itemgetter(1)
vals = [ g(i) for i in l ]
print '      values:', vals
print '      sorted:', sorted(l, key=g)
```

除了序列外，元素获取方法还适用于映射。

```
$ python operator_itemgetter.py
```

```
Dictionaries: [{'val': 0}, {'val': -1}, {'val': -2}, {'val': -3}]
      values: [0, -1, -2, -3]
      sorted: [{'val': -3}, {'val': -2}, {'val': -1}, {'val': 0}]

Tuples      : [(0, 0), (1, -2), (2, -4), (3, -6)]
      values: [0, -2, -4, -6]
      sorted: [(3, -6), (2, -4), (1, -2), (0, 0)]
```

3.3.7 结合操作符和定制类

`operator` 模块中的函数通过相应操作的标准 Python 接口完成工作，所以它们不仅适用于内置类型，还适用于用户定义的类。

```
from operator import *

class MyObj(object):
    """Example for operator overloading"""
    def __init__(self, val):
        super(MyObj, self).__init__()
        self.val = val
    return
```

```

def __str__(self):
    return 'MyObj(%s)' % self.val
def __lt__(self, other):
    """compare for less-than"""
    print 'Testing %s < %s' % (self, other)
    return self.val < other.val
def __add__(self, other):
    """add values"""
    print 'Adding %s + %s' % (self, other)
    return MyObj(self.val + other.val)
a = MyObj(1)
b = MyObj(2)

print 'Comparison:'
print lt(a, b)

print '\nArithmetic:'
print add(a, b)

```

要全面了解每个操作符使用的所有特殊方法，请参考 Python 参考指南。

```
$ python operator_classes.py
```

```

Comparison:
Testing MyObj(1) < MyObj(2)
True

Arithmetic:
Adding MyObj(1) + MyObj(2)
MyObj(3)

```

3.3.8 类型检查

operator 模块还包含一些函数来测试映射、数字和序列类型的 API 兼容性。

```

from operator import *

class NoType(object):
    """Supports none of the type APIs"""

class MultiType(object):
    """Supports multiple type APIs"""
    def __len__(self):
        return 0
    def __getitem__(self, name):
        return 'mapping'
    def __int__(self):
        return 0

```



```

o = NoType()
t = MultiType()

for func in (isMappingType, isNumberType, isSequenceType):
    print '%s(o):' % func.__name__, func(o)
    print '%s(t):' % func.__name__, func(t)

```

这些测试并不完善，因为接口没有严格定义，不过通过这些测试，确实能让我们对支持哪些功能有所了解。

```
$ python operator_typechecking.py
```

```

isMappingType(o): False
isMappingType(t): True
isNumberType(o): False
isNumberType(t): True
isSequenceType(o): False
isSequenceType(t): True

```

参见：

operator (<http://docs.python.org/lib/module-operator.html>) 这个模块的标准库文档。

functools (3.1 节) 函数编程工具，包括 total_ordering() 修饰符用于为类添加富比较方法。

itertools (3.2 节) 迭代器操作。

abc (18.2 节) abc 模块包含一些抽象基类，为集合类型定义了 API。

3.4 contextlib——上下文管理器工具

作用：创建和处理上下文管理器的工具。

Python 版本：2.5 及以后版本

contextlib 模块包含一些工具，用于处理上下文管理器和 with 语句。

注意：上下文管理器要与 with 语句关联。由于官方认为 with 是 Python 2.6 的一部分，因此在 Python 2.5 中使用 contextlib 之前要从 `__future__` 将其导入。

3.4.1 上下文管理器 API

上下文管理器 (context manager) 要负责一个代码块中的资源，可能在进入代码块时创建资源，然后在退出代码块时清理这个资源。例如，文件支持上下文管理器 API，可以很容易地确保完成文件读写后关闭文件。

```

with open('/tmp/pymotw.txt', 'wt') as f:
    f.write('contents go here')
# file is automatically closed

```

上下文管理器由 with 语句启用，这个 API 包括两个方法。当执行流进入 with 中的代码块

时会运行 `__enter__()` 方法。它会返回一个对象，在这个上下文中使用。当执行流离开 `with` 块时，则调用这个上下文管理器的 `__exit__()` 方法来清理所使用的资源。

```
class Context(object):
    def __init__(self):
        print '__init__()'
    def __enter__(self):
        print '__enter__()'
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print '__exit__()'

with Context():
    print 'Doing work in the context'
```

结合上下文管理器与 `with` 语句是 `try:finally` 块的一种更紧凑的写法，因为上下文管理器的 `__exit__()` 方法总会调用，即使在产生异常的情况下也是如此。

```
$ python contextlib_api.py
```

```
__init__()
__enter__()
Doing work in the context
__exit__()
```

如果在 `with` 语句的 `as` 子句中指定了名称，`__enter__()` 方法可以返回与这个名称相关联的任何对象。在这个例子中，`Context` 会返回使用了上下文的对象。

```
class WithinContext(object):
    def __init__(self, context):
        print 'WithinContext.__init__(%s)' % context
    def do_something(self):
        print 'WithinContext.do_something()'
    def __del__(self):
        print 'WithinContext.__del__'

class Context(object):
    def __init__(self):
        print 'Context.__init__()'
    def __enter__(self):
        print 'Context.__enter__()'
        return WithinContext(self)
    def __exit__(self, exc_type, exc_val, exc_tb):
        print 'Context.__exit__()'

with Context() as c:
    c.do_something()
```

与变量 `c` 关联的值是 `__enter__()` 返回的对象，这不一定是 `with` 语句中创建的 `Context`

实例。

```
$ python contextlib_api_other_object.py

Context.__init__()
Context.__enter__()
WithinContext.__init__(<__main__.Context object at 0x100d98a10>)
WithinContext.do_something()
Context.__exit__()
WithinContext.__del__
```

`__exit__()` 方法接收一些参数, 其中包含 `with` 块中产生的异常的详细信息。

```
class Context(object):
    def __init__(self, handle_error):
        print '__init__ (%s)' % handle_error
        self.handle_error = handle_error
    def __enter__(self):
        print '__enter__()'
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print '__exit__()'
        print ' exc_type =', exc_type
        print ' exc_val  =', exc_val
        print ' exc_tb   =', exc_tb
        return self.handle_error

with Context(True):
    raise RuntimeError('error message handled')

print

with Context(False):
    raise RuntimeError('error message propagated')
```

如果上下文管理器可以处理这个异常, `__exit__()` 应当返回一个 `true` 值来指示不需要传播这个异常。如果返回 `false`, 就会导致 `__exit__()` 返回后重新抛出这个异常。

```
$ python contextlib_api_error.py

__init__(True)
__enter__()
__exit__()
exc_type = <type 'exceptions.RuntimeError'>
exc_val  = error message handled.
exc_tb   = <traceback object at 0x100da52d8>

__init__(False)
__enter__()
```

```

__exit__()
exc_type = <type 'exceptions.RuntimeError'>
exc_val  = error message propagated
exc_tb   = <traceback object at 0x100da5368>
Traceback (most recent call last):
  File "contextlib_api_error.py", line 33, in <module>
    raise RuntimeError('error message propagated')
RuntimeError: error message propagated

```

3.4.2 从生成器到上下文管理器

采用传统方式创建上下文管理器，即编写一个包含 `__enter__()` 和 `__exit__()` 方法的类，这并不难。不过有些时候，对于很少的上下文来说，完全编写所有代码会是额外的负担。在这些情况下，可以使用 `contextmanager()` 修饰符将一个生成器函数转换为上下文管理器。

```

import contextlib

@contextlib.contextmanager
def make_context():
    print 'entering'
    try:
        yield {}
    except RuntimeError, err:
        print 'ERROR:', err
    finally:
        print 'exiting'

print 'Normal:'
with make_context() as value:
    print 'inside with statement:', value

print '\nHandled error:'
with make_context() as value:
    raise RuntimeError('showing example of handling an error')

print '\nUnhandled error:'
with make_context() as value:
    raise ValueError('this exception is not handled')

```

生成器要初始化上下文，用 `yield` 生成一次值，然后清理上下文。所生成的值（如果有）会绑定到 `with` 语句 `as` 子句中的变量。`with` 块中的异常会在生成器中重新抛出，使之在生成器中得到处理。

```
$ python contextlib_contextmanager.py
```

```

Normal:
    entering

```

```

    inside with statement: {}
    exiting

Handled error:
    entering
    ERROR: showing example of handling an error
    exiting

Unhandled error:
    entering
    exiting
Traceback (most recent call last):
  File "contextlib_contextmanager.py", line 34, in <module>
    raise ValueError('this exception is not handled')
ValueError: this exception is not handled

```

3.4.3 嵌套上下文

有些情况下，必须同时管理多个上下文（例如在输入和输出文件句柄之间复制数据）。为此可以嵌套 with 语句，不过，如果外部上下文并不需要单独的块，嵌套 with 语句就只会增加缩进层次而不会提供任何实际的好处。使用 nested() 可以利用一条 with 语句实现上下文嵌套。

```

import contextlib

@contextlib.contextmanager
def make_context(name):
    print 'entering:', name
    yield name
    print 'exiting :', name

with contextlib.nested(make_context('A'),
                      make_context('B')) as (A, B):
    print 'inside with statement:', A, B

```

程序执行时会按其进入上下文的逆序离开上下文。

```
$ python contextlib_nested.py
```

```

entering: A
entering: B
inside with statement: A B
exiting : B
exiting : A

```

在 Python 2.7 及以后版本中废弃了 nested()，因为在这些版本中 with 语句直接支持嵌套。

```
import contextlib
```

```
@contextlib.contextmanager
def make_context(name):
    print 'entering:', name
    yield name
    print 'exiting :', name

with make_context('A') as A, make_context('B') as B:
    print 'inside with statement:', A, B
```

每个上下文管理器与 `as` 子句（可选）之间用一个逗号（,）分隔。其效果类似于使用 `nested()`，不过避免了 `nested()` 不能正确实现的有关错误处理的一些边界情况。

```
$ python contextlib_nested_with.py
```

```
entering: A
entering: B
inside with statement: A B
exiting : B
exiting : A
```

3.4.4 关闭打开的句柄

`file` 类直接支持上下文管理器 API，不过表示打开句柄的另外一些对象并不支持这个 API。`contextlib` 的标准库文档中给出的例子是一个由 `urllib.urlopen()` 返回的对象。还有另外一些遗留类，它们使用 `close()` 方法而不支持上下文管理器 API。为了确保关闭句柄，需要使用 `closing()` 为它创建一个上下文管理器。

```
import contextlib

class Door(object):
    def __init__(self):
        print ' __init__()'
    def close(self):
        print ' close()'

print 'Normal Example:'
with contextlib.closing(Door()) as door:
    print ' inside with statement'

print '\nError handling example:'
try:
    with contextlib.closing(Door()) as door:
        print ' raising from inside with statement'
        raise RuntimeError('error message')
except Exception, err:
    print ' Had an error:', err
```



不论 with 块中是否有一个错误，这个句柄都会关闭。

```
$ python contextlib_closing.py
```

Normal Example:

```
__init__()  
inside with statement  
close()
```

Error handling example:

```
__init__()  
raising from inside with statement  
close()  
Had an error: error message
```

参见:

contextlib (<http://docs.python.org/library/contextlib.html>) 这个模块的标准库文档。

PEP 343 (<http://www.python.org/dev/peps/pep-0343>) with 语句。

Context Manager Types (<http://docs.python.org/library/stdtypes.html#type-contextmanager>) 标准库文档中关于上下文管理器 API 的描述。

With Statement Context Managers(<http://docs.python.org/reference/datamodel.html#context-managers>) Python 参考指南中关于上下文管理器 API 的描述。



第 ④ 章

日期和时间

不同于 `int`、`float` 和 `str`，Python 没有包含对应日期和时间的内置类型，不过提供了 3 个相应模块，可以采用多种表示管理日期和时间值。

- `time` 模块由底层 C 库提供与时间相关的函数。它包含一些函数用于获取时钟时间和处理器运行时间，还提供了基本解析和字符串格式化工具。
- `datetime` 模块为日期、时间以及日期时间值提供了一个更高层接口。`datetime` 中的类支持算术、比较和时区配置。
- `calendar` 模块可以创建周、月和年的格式化表示。它还可以用来计算重复事件、给定日期是星期几，以及其他基于日历的值。

4.1 `time`——时钟时间

作用：管理时钟时间的函数。

Python 版本：1.4 及以后版本

`time` 模块提供了一些用于管理日期和时间的 C 库函数。由于它绑定到底层 C 实现，一些细节（如纪元开始时间和支持的最大日期值）会特定于具体的平台。要全面了解有关的详细信息，请参考库文档。

4.1.1 壁挂钟时间

`time` 模块的核心函数之一是 `time()`，它会把从纪元开始以来的秒数作为一个浮点值返回。

```
import time
```

```
print 'The time is:', time.time()
```

尽管这个值总是一个浮点数，但具体的精度依赖于具体的平台。

```
$ python time_time.py
```

```
The time is: 1291499267.33
```

浮点数表示对于存储或比较日期很有用，但是对于生成人类可读的表示就有些差强人意了。要记录或打印时间，`ctime()` 可能更有用。

```
import time

print 'The time is      :', time.ctime()
later = time.time() + 15
print '15 secs from now :', time.ctime(later)
```

这个例子中的第二个 print 语句显示了如何使用 ctime() 来格式化当前时间以外的另一个时间值。

```
$ python time_ctime.py

The time is      : Sat Dec  4 16:47:47 2010
15 secs from now : Sat Dec  4 16:48:02 2010
```

4.1.2 处理器时钟时间

time() 返回的是一个壁挂钟时间，而 clock() 会返回处理器时钟时间。clock() 返回的值应当用于性能测试、基准测试等，因为它们反映了程序使用的实际时间，可能比 time() 返回的值更精确。

```
import hashlib
import time

# Data to use to calculate md5 checksums
data = open(__file__, 'rt').read()

for i in range(5):
    h = hashlib.shal()
    print time.ctime(), ': %0.3f %0.3f' % (time.time(), time.clock())
    for i in range(300000):
        h.update(data)
    cksum = h.digest()
```

在这个例子中，每次循环迭代时会打印格式化的 ctime() 时间，以及 time() 和 clock() 返回的浮点数值。

注意：如果希望在你的系统上运行这个例子，可能必须向内循环增加更多周期，或者处理更大量的数据才能真正看到时间的差异。

```
$ python time_clock.py

Sat Dec  4 16:47:47 2010 : 1291499267.446 0.028
Sat Dec  4 16:47:48 2010 : 1291499268.844 1.413
Sat Dec  4 16:47:50 2010 : 1291499270.247 2.794
Sat Dec  4 16:47:51 2010 : 1291499271.658 4.171
Sat Dec  4 16:47:53 2010 : 1291499273.128 5.549
```

一般地，如果程序什么也没有做，处理器时钟不会“滴答”（tick）。

```
import time

for i in range(6, 1, -1):
    print '%s %0.2f %0.2f' % (time.ctime(),
                              time.time(),
                              time.clock())

    print 'Sleeping', i
    time.sleep(i)
```

在这个例子中，循环几乎不做什么工作，每次迭代后都会睡眠。即使应用睡眠，time() 值也会增加，但是 clock() 值不会增加。

```
$ python time_clock_sleep.py

Sat Dec  4 16:47:54 2010 1291499274.65 0.03
Sleeping 6
Sat Dec  4 16:48:00 2010 1291499280.65 0.03
Sleeping 5
Sat Dec  4 16:48:05 2010 1291499285.65 0.03
Sleeping 4
Sat Dec  4 16:48:09 2010 1291499289.66 0.03
Sleeping 3
Sat Dec  4 16:48:12 2010 1291499292.66 0.03
Sleeping 2
```

调用 sleep() 会从当前线程交出控制，要求它等待系统将其再次唤醒。如果程序只有一个线程，这实际上就会阻塞应用，什么也不做。

4.1.3 时间组成

有些情况下需要将时间存储为过去了多少秒（秒数），但是还有一些情况，程序需要访问一个日期的各个字段（年、月等）。time 模块定义了 struct_time 来维护日期和时间值，其中分开存储各个组成部分，以便于访问。很多函数都要处理 struct_time 值而不是浮点数值。

```
import time

def show_struct(s):
    print 'tm_year:', s.tm_year
    print 'tm_mon:', s.tm_mon
    print 'tm_mday:', s.tm_mday
    print 'tm_hour:', s.tm_hour
    print 'tm_min:', s.tm_min
    print 'tm_sec:', s.tm_sec
    print 'tm_wday:', s.tm_wday
    print 'tm_yday:', s.tm_yday
```

```

print 'tm_isdst:', s.tm_isdst
print 'gmtime:'
show_struct(time.gmtime())
print '\nlocaltime:'
show_struct(time.localtime())
print '\nmktime:', time.mktime(time.localtime())

```

gmtime() 函数以 UTC 格式返回当前时间。localtime() 会返回应用了当前时区的当前时间。mktime() 取一个 struct_time 实例，将它转换为浮点数表示。

```
$ python time_struct.py
```

```

gmtime:
tm_year : 2010
tm_mon  : 12
tm_mday : 4
tm_hour : 21
tm_min  : 48
tm_sec  : 14
tm_wday : 5
tm_yday : 338
tm_isdst: 0

```

```

localtime:
tm_year : 2010
tm_mon  : 12
tm_mday : 4
tm_hour : 16
tm_min  : 48
tm_sec  : 14
tm_wday : 5
tm_yday : 338
tm_isdst: 0

```

```
mktime: 1291499294.0
```

4.1.4 处理时区

确定当前时间的函数有一个前提，即已经设置了时区，这可能由程序设置，也可以使用系统的默认时区。修改时区不会改变具体的时间，只是会改变表示时间的方式。

要改变时区，需要设置环境变量 TZ，然后调用 tzset()。设置时区时可以指定很多细节，甚至细致到日光节省时间的开始和结束时间。不过，通常更容易的做法是使用时区名，并由底层库推导出其他信息。

下面这个示例程序将修改时区值，并说明这些改变对 time 模块中的其他设置有什么影响。

```

import time
import os

```

```

def show_zone_info():
    print ' TZ      :', os.environ.get('TZ', '(not set)')
    print ' tzname:', time.tzname
    print ' Zone   : %d (%d)' % (time.timezone,
                                (time.timezone / 3600))
    print ' DST    :', time.daylight
    print ' Time   :', time.ctime()
    print

print 'Default : '
show_zone_info()

ZONES = [ 'GMT',
          'Europe/Amsterdam',
          ]

for zone in ZONES:
    os.environ['TZ'] = zone
    time.tzset()
    print zone, ':'
    show_zone_info()

```

用于这些例子的系统默认时区是 US/Eastern。例程中的其他时区改变了 tzname、daylight 标志和时区偏移值。

```
$ python time_timezone.py
```

```

Default :
TZ      : (not set)
tzname: ('EST', 'EDT')
Zone   : 18000 (5)
DST    : 1
Time   : Sat Dec  4 16:48:14 2010

```

```

GMT :
TZ      : GMT
tzname: ('GMT', 'GMT')
Zone   : 0 (0)
DST    : 0
Time   : Sat Dec  4 21:48:14 2010

```

```

Europe/Amsterdam :
TZ      : Europe/Amsterdam
tzname: ('CET', 'CEST')
Zone   : -3600 (-1)
DST    : 1
Time   : Sat Dec  4 22:48:15 2010

```



4.1.5 解析和格式化时间

time 模块提供了两个函数 `strptime()` 和 `strftime()`，可以在 `struct_time` 和时间值字符串表示之间转换。模块提供了大量格式化指令来支持以不同方式输入和输出。所有这些格式化指令的完整列表参见 time 模块的库文档。

下面的这个例子将当前时间从一个字符串转换为一个 `struct_time` 实例，然后再转换回一个字符串。

```
import time

def show_struct(s):
    print ' tm_year :', s.tm_year
    print ' tm_mon  :', s.tm_mon
    print ' tm_mday  :', s.tm_mday
    print ' tm_hour  :', s.tm_hour
    print ' tm_min   :', s.tm_min
    print ' tm_sec   :', s.tm_sec
    print ' tm_wday  :', s.tm_wday
    print ' tm_yday  :', s.tm_yday
    print ' tm_isdst:', s.tm_isdst

now = time.ctime()
print 'Now:', now
parsed = time.strptime(now)
print '\nParsed:'
show_struct(parsed)

print '\nFormatted:', time.strftime("%a %b %d %H:%M:%S %Y", parsed)
```

输出字符串与输入字符串并不完全相同，因为日期前面加了一个前缀 0（由“4”变为“04”）。

```
$ python time_strptime.py
```

```
Now: Sat Dec  4 16:48:14 2010
```

```
Parsed:
 tm_year : 2010
 tm_mon  : 12
 tm_mday : 4
 tm_hour : 16
 tm_min  : 48
 tm_sec  : 14
 tm_wday : 5
 tm_yday : 338
 tm_isdst: -1
```

```
Formatted: Sat Dec 04 16:48:14 2010
```



参见:

time (<http://docs.python.org/lib/module-time.html>) 这个模块的标准库文档。

datetime (4.2 节) datetime 模块包含一些类用于完成关于日期和时间的计算。

calendar (4.3 节) 处理更高级的日期函数, 来生成日历或计算重复事件。

4.2 datetime——日期和时间值管理

作用: datetime 模块包含一些函数和类, 用于完成日期和时间解析、格式化和算术运算。

Python 版本: 2.3 及以后版本

datetime 包含一些用于处理日期和时间的函数和类, 这些函数和类可以单独使用, 也可以结合使用。

4.2.1 时间

时间值用 time 类表示。time 实例包含 hour、minute、second 和 microsecond 属性, 还可以包含时区信息。

```
import datetime

t = datetime.time(1, 2, 3)
print t
print 'hour      :', t.hour
print 'minute    :', t.minute
print 'second     :', t.second
print 'microsecond:', t.microsecond
print 'tzinfo     :', t.tzinfo
```

初始化 time 实例的参数是可选的, 不过默认值 0 往往不会是正确的设置。

```
$ python datetime_time.py
```

```
01:02:03
hour      : 1
minute    : 2
second     : 3
microsecond: 0
tzinfo     : None
```

time 实例只包含时间值, 而不包含与时间关联的日期值。

```
import datetime

print 'Earliest  :', datetime.time.min
print 'Latest    :', datetime.time.max
print 'Resolution:', datetime.time.resolution
```

min 和 max 类属性可以反映一天中的合法时间范围。




```
$ python datetime_time_minmax.py
```

```
Earliest : 00:00:00
Latest   : 23:59:59.999999
Resolution: 0:00:00.000001
```

time 的分辨率限制为整毫秒值。

```
import datetime
```

```
for m in [ 1, 0, 0.1, 0.6 ]:
    try:
        print '%02.1f :' % m, datetime.time(0, 0, 0, microsecond=m)
    except TypeError, err:
        print 'ERROR:', err
```

处理浮点数值的方式取决于 Python 的版本。版本 2.7 会产生一个 `TypeError`，而更早的版本会生成一个 `DeprecationWarning`，并把浮点数转换为一个整数。

```
$ python2.7 datetime_time_resolution.py
```

```
1.0 : 00:00:00.000001
0.0 : 00:00:00
0.1 : ERROR: integer argument expected, got float
0.6 : ERROR: integer argument expected, got float
```

```
$ python2.6 datetime_time_resolution.py
```

```
1.0 : 00:00:00.000001
0.0 : 00:00:00
datetime_time_resolution.py:16: DeprecationWarning: integer argument
expected, got float
print '%02.1f :' % m, datetime.time(0, 0, 0, microsecond=m)
0.1 : 00:00:00
0.6 : 00:00:00
```

4.2.2 日期

日历日期值用 `date` 类表示。`date` 实例包含 `year`、`month` 和 `day` 属性。使用 `today()` 类方法很容易创建一个表示当前日期的日期实例。

```
import datetime
```

```
today = datetime.date.today()
print today
print 'ctime :', today.ctime()
tt = today.timetuple()
print 'tuple : tm_year =', tt.tm_year
```

```

print 'tm_mon =', tt.tm_mon
print 'tm_mday =', tt.tm_mday
print 'tm_hour =', tt.tm_hour
print 'tm_min =', tt.tm_min
print 'tm_sec =', tt.tm_sec
print 'tm_wday =', tt.tm_wday
print 'tm_yday =', tt.tm_yday
print 'tm_isdst =', tt.tm_isdst
print 'ordinal:', today.toordinal()
print 'Year :', today.year
print 'Mon :', today.month
print 'Day :', today.day

```

下面这个例子采用多种不同格式打印当前日期。

```
$ python datetime_date.py
```

```

2010-11-27
ctime : Sat Nov 27 00:00:00 2010
tuple : tm_year = 2010
        tm_mon  = 11
        tm_mday = 27
        tm_hour = 0
        tm_min  = 0
        tm_sec  = 0
        tm_wday = 5
        tm_yday = 331
        tm_isdst = -1
ordinal: 734103
Year   : 2010
Mon    : 11
Day    : 27

```

还有一些类方法可以由 POSIX 时间戳或表示 Gregorian 日历中日期值的整数（第 1 年的 1 月 1 日对应的整数为 1，以后每天对应的值相应增 1）来创建 date 实例。

```

import datetime
import time

o = 733114
print 'o :', o
print 'fromordinal(o) :', datetime.date.fromordinal(o)

t = time.time()
print 't :', t
print 'fromtimestamp(t) :', datetime.date.fromtimestamp(t)

```

这个例子表明 fromordinal() 和 fromtimestamp() 使用了不同的值类型。

```
$ python datetime_date_fromordinal.py
```

```
o           : 733114
fromordinal(o) : 2008-03-13
t           : 1290874810.14
fromtimestamp(t): 2010-11-27
```

与 `time` 类似，可以使用 `min` 和 `max` 属性确定所支持的日期值范围。

```
import datetime
```

```
print 'Earliest  :', datetime.date.min
print 'Latest    :', datetime.date.max
print 'Resolution:', datetime.date.resolution
```

日期的分辨率为整天。

```
$ python datetime_date_minmax.py
```

```
Earliest : 0001-01-01
Latest   : 9999-12-31
Resolution: 1 day, 0:00:00
```

创建新的 `date` 实例还有一种方法，可以使用一个现有 `date` 的 `replace()` 方法来创建。

```
import datetime
```

```
d1 = datetime.date(2008, 3, 29)
print 'd1:', d1.ctime()
d2 = d1.replace(year=2009)
print 'd2:', d2.ctime()
```

下面这个例子会改变年，但日和月保持不变。

```
$ python datetime_date_replace.py
```

```
d1: Sat Mar 29 00:00:00 2008
d2: Sun Mar 29 00:00:00 2009
```

4.2.3 timedelta

通过对两个 `datetime` 对象使用算术运算，或者结合使用一个 `datetime` 和一个 `timedelta`，可以计算出将来和过去的一些日期。将两个日期相减可以生成一个 `timedelta`，还可以对某个日期增加或减去一个 `timedelta` 来生成另一个日期。`timedelta` 的内部值按日、秒和毫秒存储。

```
import datetime
```

```
print "microseconds:", datetime.timedelta(microseconds=1)
print "milliseconds:", datetime.timedelta(milliseconds=1)
print "seconds      :", datetime.timedelta(seconds=1)
print "minutes      :", datetime.timedelta(minutes=1)
```

```

print "hours      :", datetime.timedelta(hours=1)
print "days      :", datetime.timedelta(days=1)
print "weeks      :", datetime.timedelta(weeks=1)

```

传入构造函数的中间值会转换为日、秒和毫秒。

```
$ python datetime_timedelta.py
```

```

microseconds: 0:00:00.000001
milliseconds: 0:00:00.001000
seconds      : 0:00:01
minutes      : 0:01:00
hours        : 1:00:00
days        : 1 day, 0:00:00
weeks        : 7 days, 0:00:00

```

一个 `timedelta` 的完整时间段可以使用 `total_seconds()` 得到，作为一个秒数返回。

```

import datetime

for delta in [datetime.timedelta(microseconds=1),
              datetime.timedelta(milliseconds=1),
              datetime.timedelta(seconds=1),
              datetime.timedelta(minutes=1),
              datetime.timedelta(hours=1),
              datetime.timedelta(days=1),
              datetime.timedelta(weeks=1),
              ]:
    print '%15s = %s seconds' % (delta, delta.total_seconds())

```

返回值是一个浮点数，因为有些时间段不到 1 秒。

```
$ python datetime_timedelta_total_seconds.py
```

```

0:00:00.000001 = 1e-06 seconds
0:00:00.001000 = 0.001 seconds
0:00:01        = 1.0 seconds
0:01:00        = 60.0 seconds
1:00:00        = 3600.0 seconds
1 day, 0:00:00 = 86400.0 seconds
7 days, 0:00:00 = 604800.0 seconds

```

4.2.4 日期算术运算

日期算术运算使用标准算术操作符来完成。

```

import datetime

today = datetime.date.today()
print 'Today      :', today

```



```

one_day = datetime.timedelta(days=1)
print 'One day :', one_day

yesterday = today - one_day
print 'Yesterday:', yesterday
tomorrow = today + one_day
print 'Tomorrow :', tomorrow

print
print 'tomorrow - yesterday:', tomorrow - yesterday
print 'yesterday - tomorrow:', yesterday - tomorrow

```

这个处理日期对象的例子展示了使用 `timedelta` 对象来计算新日期，以及将日期实例相减来生成 `timedelta`（包括一个负差异值）。

```

$ python datetime_date_math.py

Today      : 2010-11-27
One day    : 1 day, 0:00:00
Yesterday: 2010-11-26
Tomorrow   : 2010-11-28

tomorrow - yesterday: 2 days, 0:00:00
yesterday - tomorrow: -2 days, 0:00:00

```

4.2.5 比较值

日期和时间值都可以使用标准比较操作符来比较，确定哪一个在前或在后。

```

import datetime
import time

print 'Times:'
t1 = datetime.time(12, 55, 0)
print ' t1:', t1
t2 = datetime.time(13, 5, 0)
print ' t2:', t2
print ' t1 < t2:', t1 < t2

print
print 'Dates:'
d1 = datetime.date.today()
print ' d1:', d1
d2 = datetime.date.today() + datetime.timedelta(days=1)
print ' d2:', d2
print ' d1 > d2:', d1 > d2

```

所有比较操作符都得到支持。



```
$ python datetime_comparing.py
```

```
Times:
  t1: 12:55:00
  t2: 13:05:00
  t1 < t2: True
```

```
Dates:
  d1: 2010-11-27
  d2: 2010-11-28
  d1 > d2: False
```

4.2.6 结合日期和时间

使用 `datetime` 类可以存储由日期和时间分量构成的值。类似于 `date`，可以使用很多便利的类方法由其他常用值创建 `datetime` 实例。

```
import datetime

print 'Now      :', datetime.datetime.now()
print 'Today    :', datetime.datetime.today()
print 'UTC Now:', datetime.datetime.utcnow()
print

FIELDS = [ 'year', 'month', 'day',
            'hour', 'minute', 'second', 'microsecond',
          ]
```

```
d = datetime.datetime.now()
for attr in FIELDS:
    print '%15s: %s' % (attr, getattr(d, attr))
```

可以想见，`datetime` 实例包含 `date` 和 `time` 对象的所有属性。

```
$ python datetime_datetime.py
```

```
Now      : 2010-11-27 11:20:10.479880
Today    : 2010-11-27 11:20:10.481494
UTC Now  : 2010-11-27 16:20:10.481521
```

```
      year: 2010
      month: 11
       day: 27
      hour: 11
     minute: 20
      second: 10
microsecond: 481752
```

与 `date` 类似，`datetime` 提供了方便的类方法来创建新实例。它还包括 `fromordinal()` 和



fromtimestamp()。

```
import datetime

t = datetime.time(1, 2, 3)
print 't:', t

d = datetime.date.today()
print 'd:', d

dt = datetime.datetime.combine(d, t)
print 'dt:', dt
```

利用 combine(), 可以由一个 date 实例和一个 time 实例创建一个 datetime 实例。

```
$ python datetime_datetime_combine.py
```

```
t: 01:02:03
d: 2010-11-27
dt: 2010-11-27 01:02:03
```

4.2.7 格式化和解析

datetime 对象的默认字符串表示使用 ISO-8601 格式 (YYYY-MM-DDTHH:MM:SS.mmmmmmm)。可以使用 strftime() 生成其他格式。

```
import datetime

format = "%a %b %d %H:%M:%S %Y"

today = datetime.datetime.today()
print 'ISO      :', today

s = today.strftime(format)
print 'strftime:', s

d = datetime.datetime.strptime(s, format)
print 'strptime:', d.strftime(format)
```

使用 datetime.strptime() 可以将格式化的字符串转换为 datetime 实例。

```
$ python datetime_datetime_strptime.py
```

```
ISO      : 2010-11-27 11:20:10.571582
strftime: Sat Nov 27 11:20:10 2010
strptime: Sat Nov 27 11:20:10 2010
```

4.2.8 时区

在 datetime 中, 时区由 tzinfo 的子类表示。由于 tzinfo 是一个抽象基类, 实际使用时, 应

用需要定义它的一个子类，并提供一些方法的适当的表示。遗憾的是，`datetime` 未包含任何可供使用的具体实现（不过文档提供了一些示例实现）。可以参考标准库文档页面，其中有一些使用固定偏移的例子，还可以从中了解一个有关 DST 的类以及创建定制时区类的更多详细信息。`pytz` 也是一个了解时区实现细节的很好的资源。

参见：

`datetime` (<http://docs.python.org/lib/module-datetime.html>) 这个模块的标准库文档。

`calendar` (4.3 节) `calendar` 模块。

`time` (4.1 节) `time` 模块。

`dateutil` (<http://labix.org/python-dateutil>) Labix 的 `dateutil` 为 `datetime` 模块扩展了一些额外特性。

Wikipedia: Proleptic Gregorian calendar (http://en.wikipedia.org/wiki/Proleptic_Gregorian_calendar) 对 Gregorian 日历系统的一个描述。

`pytz` (<http://pytz.sourceforge.net/>) World Time Zone 数据库。

ISO 8601 (http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/date_and_time_format.htm) 日期和时间数值表示的标准。

4.3 calendar——处理日期

作用：`calendar` 模块实现了一些类来处理日期，管理面向年、月和周的值。

Python 版本：1.4 版本，2.5 中做了更新

`calendar` 模块定义了 `Calendar` 类，其中封装了一些值的计算，如给定的一个月或一年中的周几。另外，`TextCalendar` 和 `HTMLCalendar` 类可以生成经过预格式化的输出。

4.3.1 格式化示例

`prmonth()` 方法是一个简单的函数，可以生成一个月的格式化文本输出。

```
import calendar
```

```
c = calendar.TextCalendar(calendar.SUNDAY)
c.prmonth(2011, 7)
```

这个例子按照美国的惯例，将 `TextCalendar` 配置为一周从星期日开始。默认会使用欧洲惯例，即一周从星期一开始。

输出如下：

```
$ python calendar_textcalendar.py
```

```

    July 2011
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
```



```

10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

```

利用 `HTMLCalendar` 和 `formatmonth()` 可以生成一个类似的 HTML 表格。显示的输出看起来与纯文本的版本大致是一样的，不过会用 HTML 标记包围。各个表单元格有一个类属性对应星期几，所以可以通过 CSS 指定 HTML 样式。

要使用可用默认格式之外的某种格式生成输出，可以使用 `calendar` 计算日期，并把这些值组织为周和月，然后迭代处理结果。对于这个任务，`Calendar` 的 `weekheader()`、`monthcalendar()` 和 `yeardays2calendar()` 方法尤其有用。

调用 `yeardays2calendar()` 会生成一个由“月栏”列表构成的序列。每个列表包含一些月，每个月是一个周列表。周是元组列表，元组则由日编号（1~31）和星期几（0~6）构成。当月以外的日编号为 0。

```

import calendar
import pprint

cal = calendar.Calendar(calendar.SUNDAY)

cal_data = cal.yeardays2calendar(2011, 3)
print 'len(cal_data)      :', len(cal_data)

top_months = cal_data[0]
print 'len(top_months)    :', len(top_months)

first_month = top_months[0]
print 'len(first_month)   :', len(first_month)

print 'first_month:'
pprint.pprint(first_month)

```

调用 `yeardays2calendar(2011, 3)` 会返回 2011 年的数据，按每栏 3 个月组织。

```
$ python calendar_yeardays2calendar.py
```

```

len(cal_data)      : 4
len(top_months)    : 3
len(first_month)   : 6
first_month:
[[ (0, 6), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 5)],
  [(2, 6), (3, 0), (4, 1), (5, 2), (6, 3), (7, 4), (8, 5)],
  [(9, 6), (10, 0), (11, 1), (12, 2), (13, 3), (14, 4), (15, 5)],
  [(16, 6), (17, 0), (18, 1), (19, 2), (20, 3), (21, 4), (22, 5)],
  [(23, 6), (24, 0), (25, 1), (26, 2), (27, 3), (28, 4), (29, 5)],
  [(30, 6), (31, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]]

```

这等价于 `formatyear()` 使用的数据。

```
import calendar
```

```
cal = calendar.TextCalendar(calendar.SUNDAY)
```

```
print cal.formatyear(2011, 2, 1, 1, 3)
```

对于相同的参数，`formatyear()` 会生成以下输出。

```
$ python calendar_formatyear.py
```

2011

January							February							March							
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	
						1			1	2	3	4	5				1	2	3	4	5
2	3	4	5	6	7	8	6	7	8	9	10	11	12	6	7	8	9	10	11	12	
9	10	11	12	13	14	15	13	14	15	16	17	18	19	13	14	15	16	17	18	19	
16	17	18	19	20	21	22	20	21	22	23	24	25	26	20	21	22	23	24	25	26	
23	24	25	26	27	28	29	27	28	27	28	29	30	31								
30	31																				

April							May							June						
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
					1	2	1	2	3	4	5	6	7				1	2	3	4
3	4	5	6	7	8	9	8	9	10	11	12	13	14	5	6	7	8	9	10	11
10	11	12	13	14	15	16	15	16	17	18	19	20	21	12	13	14	15	16	17	18
17	18	19	20	21	22	23	22	23	24	25	26	27	28	19	20	21	22	23	24	25
24	25	26	27	28	29	30	29	30	31	26	27	28	29	30						

July							August							September						
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
					1	2		1	2	3	4	5	6					1	2	3
3	4	5	6	7	8	9	7	8	9	10	11	12	13	4	5	6	7	8	9	10
10	11	12	13	14	15	16	14	15	16	17	18	19	20	11	12	13	14	15	16	17
17	18	19	20	21	22	23	21	22	23	24	25	26	27	18	19	20	21	22	23	24
24	25	26	27	28	29	30	28	29	30	31	25	26	27	28	29	30				
31																				

October							November							December						
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
						1			1	2	3	4	5					1	2	3
2	3	4	5	6	7	8	6	7	8	9	10	11	12	4	5	6	7	8	9	10
9	10	11	12	13	14	15	13	14	15	16	17	18	19	11	12	13	14	15	16	17
16	17	18	19	20	21	22	20	21	22	23	24	25	26	18	19	20	21	22	23	24
23	24	25	26	27	28	29	27	28	29	30	25	26	27	28	29	30	31			
30	31																			

`day_name`、`day_abbr`、`month_name` 和 `month_abbr` 模块属性对于生成定制格式的输出很有用（例如，在 HTML 输出中包含链接）。这些属性会针对当前本地化环境正确地自动配置。

4.3.2 计算日期

尽管 `calendar` 模块主要强调采用不同格式打印完整的日历，它还提供了另外一些函数，对采用其他方式处理日期很有用，如为一个重复事件计算日期。例如，Python Atlanta Users Group 每月的第二个星期四会召开一次会议。要计算一年中的会议日期，可以使用 `monthcalendar()` 的返回值。

```
import calendar
import pprint

pprint.pprint(calendar.monthcalendar(2011, 7))
```

有些日期的值为 0。这说明尽管这几天属于另一个月，但与给定的当前月中的几天同属一个星期。

```
$ python calendar_monthcalendar.py
```

```
[[0, 0, 0, 0, 1, 2, 3],
 [4, 5, 6, 7, 8, 9, 10],
 [11, 12, 13, 14, 15, 16, 17],
 [18, 19, 20, 21, 22, 23, 24],
 [25, 26, 27, 28, 29, 30, 31]]
```

一周中的第一天默认为星期一。可以通过调用 `setfirstweekday()` 改变这个设置，不过由于 `calendar` 模块包含一些常量来索引 `monthcalendar()` 返回的日期区间，所以在这种情况下跳过这一步会更方便。

要计算 2011 年的会议日期，假设是每个月的第二个星期四，0 值指示第一周的星期四是否包含在这个月内（或者不包含在这个月中，比如这个月从星期五开始）。

```
import calendar

# Show every month
for month in range(1, 13):

    # Compute the dates for each week that overlaps the month
    c = calendar.monthcalendar(2011, month)
    first_week = c[0]
    second_week = c[1]
    third_week = c[2]

    # If there is a Thursday in the first week, the second Thursday
    # is in the second week. Otherwise, the second Thursday must
    # be in the third week.
    if first_week[calendar.THURSDAY]:
        meeting_date = second_week[calendar.THURSDAY]
    else:
        meeting_date = third_week[calendar.THURSDAY]
```

```
print '%3s: %2s' % (calendar.month_abbrev[month], meeting_date)
```

所以，这一年的会议日程为：

```
$ python calendar_secondthursday.py
```

```
Jan: 13
Feb: 10
Mar: 10
Apr: 14
May: 12
Jun: 9
Jul: 14
Aug: 11
Sep: 8
Oct: 13
Nov: 10
Dec: 8
```

参见：

`calendar` (<http://docs.python.org/library/calendar.html>) 这个模块的标准库文档。

`time` (4.1 节) 底层时间函数。

`datetime` (4.2 节) 管理日期值，包括时间戳和时区。



第 ⑤ 章

数 学 计 算

作为一种通用的编程语言，Python 经常用来解决数学问题。它包含一些内置类型来管理整数和浮点数，这很适合完成一般应用中都可能出现的基本的数学运算。标准库包含一些模块来满足更高级的需求。

Python 的内置浮点数使用底层 double 表示。对于大多数有数学运算需求的程序来说，这已经足够精确，但是如果需要非整数值的更为精确的表示，decimal 和 fractions 模块会很有用。小数和分数值的算术运算可以保证精度，但是不如内置 float 的运算速度快。

random 模块包含一个均匀分布伪随机数生成器，还提供了一些函数来模拟很多常见的非均匀分布。

math 模块包含一些高级数学函数的快速实现，如对数和三角函数。这个模块对内置平台 C 库中常见的 IEEE 函数提供了全面的补充。

5.1 decimal——定点数和浮点数的数学运算

作用：使用定点数和浮点数的小数运算。

Python 版本：2.4 及以后版本

decimal 模块实现了定点和浮点算术运算，使用的是大多数人所熟悉的模型，而不是程序员熟悉的模式，即大多数计算机硬件实现的 IEEE 浮点数运算。Decimal 实例可以准确地表示任何数，对其上取整或下取整，还可以对有效数字个数加以限制。

5.1.1 Decimal

小数值表示为 Decimal 类的实例。构造函数取一个整数或字符串作为参数。使用浮点数创建 Decimal 之前，可以先将浮点数转换为一个字符串，使调用者能够显式地处理值的位数，倘若使用硬件浮点数表示则无法准确地表述。另外，利用类方法 from_float() 可以转换为精确的小数表示。

```
import decimal

fmt = '{0:<25} {1:<25}'
print fmt.format('Input', 'Output')
print fmt.format('-' * 25, '-' * 25)
```

```
# Integer
print fmt.format(5, decimal.Decimal(5))

# String
print fmt.format('3.14', decimal.Decimal('3.14'))

# Float
f = 0.1
print fmt.format(repr(f), decimal.Decimal(str(f)))
print fmt.format('%.23g' % f,
                  str(decimal.Decimal.from_float(f))[:25])
```

```
import decimal

a = decimal.Decimal('5.1')
b = decimal.Decimal('3.14')
c = 4
d = 3.14

print 'a      =', repr(a)
print 'b      =', repr(b)
print 'c      =', repr(c)
print 'd      =', repr(d)
print

print 'a + b =', a + b
print 'a - b =', a - b
print 'a * b =', a * b
print 'a / b =', a / b
print

print 'a + c =', a + c
print 'a - c =', a - c
print 'a * c =', a * c
print 'a / c =', a / c
print

print 'a + d =',
try:
    print a + d
except TypeError, e:
    print e
```

Decimal 运算符还接受整数参数, 不过浮点数值必须转换为 Decimal 实例。

```
$ python decimal_operators.py
```

```
a      = Decimal('5.1')
b      = Decimal('3.14')
c      = 4
d      = 3.14

a + b = 8.24
a - b = 1.96
a * b = 16.014
a / b = 1.624203821656050955414012739

a + c = 9.1
a - c = 1.1
a * c = 20.4
```



```
a / c = 1.275
```

```
a + d = unsupported operand type(s) for +: 'Decimal' and 'float'
```

除了基本算术运算，Decimal 还包括一些方法来查找以 10 为底的对数和自然对数。log10() 和 ln() 返回的值都是 Decimal 实例，所以可以与其他值一样直接在公式中使用。

5.1.3 特殊值

除了期望的数字值，Decimal 还可以表示很多特殊值，包括正负无穷大值、“不是一个数” (NaN) 和 0。

```
import decimal

for value in [ 'Infinity', 'NaN', '0' ]:
    print decimal.Decimal(value), decimal.Decimal('-' + value)
print

# Math with infinity
print 'Infinity + 1:', (decimal.Decimal('Infinity') + 1)
print '-Infinity + 1:', (decimal.Decimal('-Infinity') + 1)

# Print comparing NaN
print decimal.Decimal('NaN') == decimal.Decimal('Infinity')
print decimal.Decimal('NaN') != decimal.Decimal(1)
```

与无穷大值相加会返回另一个无穷大值。与 NaN 比较相等性总会返回 false，而比较不等性总会返回 true。与 NaN 比较大小来确定排序顺序没有明确定义，这会导致一个错误。

```
$ python decimal_special.py
```

```
Infinity -Infinity
NaN -NaN
0 -0

Infinity + 1: Infinity
-Infinity + 1: -Infinity
False
True
```

5.1.4 上下文

到目前为止，前面的例子使用的都是 decimal 模块的默认行为。还可以使用一个上下文 (context) 覆盖某些设置，如保持精度、如何完成取整、错误处理等等。上下文可以应用于一个线程中的所有 Decimal 实例，或者局部应用于一个小代码区。

当前上下文

要获取当前全局上下文，可以使用 getcontext()。


```

import decimal
import pprint

context = decimal.getcontext()

print 'Emax      =', context.Emax
print 'Emin      =', context.Emin
print 'capitals  =', context.capitals
print 'prec      =', context.prec
print 'rounding  =', context.rounding
print 'flags     ='
pprint.pprint(context.flags)
print 'traps     ='
pprint.pprint(context.traps)

```

这个示例脚本显示了 Context 的公共属性。

```
$ python decimal_getcontext.py
```

```

Emax      = 999999999
Emin      = -999999999
capitals  = 1
prec      = 28
rounding  = ROUND_HALF_EVEN
flags     =
{<class 'decimal.Clamped'>: 0,
 <class 'decimal.InvalidOperation'>: 0,
 <class 'decimal.DivisionByZero'>: 0,
 <class 'decimal.Inexact'>: 0,
 <class 'decimal.Rounded'>: 0,
 <class 'decimal.Subnormal'>: 0,
 <class 'decimal.Overflow'>: 0,
 <class 'decimal.Underflow'>: 0}
traps     =
{<class 'decimal.Clamped'>: 0,
 <class 'decimal.InvalidOperation'>: 1,
 <class 'decimal.DivisionByZero'>: 1,
 <class 'decimal.Inexact'>: 0,
 <class 'decimal.Rounded'>: 0,
 <class 'decimal.Subnormal'>: 0,
 <class 'decimal.Overflow'>: 1,
 <class 'decimal.Underflow'>: 0}

```

精度

上下文的 `prec` 属性控制着作为算术运算结果所创建的新值的精度。字面量值会按这个属性保持精度。

```
import decimal
```

```
d = decimal.Decimal('0.123456')
for i in range(4):
    decimal.getcontext().prec = i
    print i, ': ', d, d * 1
```

要改变精度，可以直接为这个属性赋一个新值。

```
$ python decimal_precision.py
```

```
0 : 0.123456 0
1 : 0.123456 0.1
2 : 0.123456 0.12
3 : 0.123456 0.123
```

取整

取整有多种选择，以保证值在所需精度范围内。

ROUND_CEILING 总是趋向无穷大向上取整。

ROUND_DOWN 总是趋向 0 取整。

ROUND_FLOOR 总是趋向负无穷大向下取整。

ROUND_HALF_DOWN 如果最后一个有效数字大于或等于 5 则朝 0 反方向取整；否则，趋向 0 取整。

ROUND_HALF_EVEN 类似于 ROUND_HALF_DOWN，不过，如果最后一个有效数字值为 5，则会检查前一位。偶数值会导致结果向下取整，奇数值导致结果向上取整。

ROUND_HALF_UP 类似于 ROUND_HALF_DOWN，不过如果最后一位有效数字为 5，值会朝 0 的反方向取整。

ROUND_UP 朝 0 的反方向取整。

ROUND_05UP 如果最后一位是 0 或 5，则朝 0 的反方向取整；否则向 0 取整。

```
import decimal
```

```
context = decimal.getcontext()
```

```
ROUNDING_MODES = [
    'ROUND_CEILING',
    'ROUND_DOWN',
    'ROUND_FLOOR',
    'ROUND_HALF_DOWN',
    'ROUND_HALF_EVEN',
    'ROUND_HALF_UP',
    'ROUND_UP',
    'ROUND_05UP',
]
header_fmt = '{:10} ' + ' '.join(['{:^8}'] * 6)
```



```

print header_fmt.format(' ',
                        '1/8 (1)', '-1/8 (1)',
                        '1/8 (2)', '-1/8 (2)',
                        '1/8 (3)', '-1/8 (3)',
                        )
for rounding_mode in ROUNDING_MODES:
    print '{0:10}'.format(rounding_mode.partition('_')[-1]),
    for precision in [ 1, 2, 3 ]:
        context.prec = precision
        context.rounding = getattr(decimal, rounding_mode)
        value = decimal.Decimal(1) / decimal.Decimal(8)
        print '{0:^8}'.format(value),
        value = decimal.Decimal(-1) / decimal.Decimal(8)
        print '{0:^8}'.format(value),
    print

```

这个程序显示了使用不同算法将同一个值取整为不同精度的效果。

```
$ python decimal_rounding.py
```

	1/8 (1)	-1/8 (1)	1/8 (2)	-1/8 (2)	1/8 (3)	-1/8 (3)
CEILING	0.2	-0.1	0.13	-0.12	0.125	-0.125
DOWN	0.1	-0.1	0.12	-0.12	0.125	-0.125
FLOOR	0.1	-0.2	0.12	-0.13	0.125	-0.125
HALF_DOWN	0.1	-0.1	0.12	-0.12	0.125	-0.125
HALF_EVEN	0.1	-0.1	0.12	-0.12	0.125	-0.125
HALF_UP	0.1	-0.1	0.13	-0.13	0.125	-0.125
UP	0.2	-0.2	0.13	-0.13	0.125	-0.125
ROUND	0.1	-0.1	0.12	-0.12	0.125	-0.125

局部上下文

使用 Python 2.5 或以后版本时, 可以使用 with 语句对一个代码块应用上下文。

```

import decimal

with decimal.localcontext() as c:
    c.prec = 2
    print 'Local precision:', c.prec
    print '3.14 / 3 =', (decimal.Decimal('3.14') / 3)
print
print 'Default precision:', decimal.getcontext().prec
print '3.14 / 3 =', (decimal.Decimal('3.14') / 3)

```

Context 支持 with 使用的上下文管理器 API, 所以这个设置只在块内应用。

```
$ python decimal_context_manager.py
```

```

Local precision: 2
3.14 / 3 = 1.0

```

```
Default precision: 28
3.14 / 3 = 1.04666666666666666666666666666667
```

各实例的上下文

上下文还可以用来构造 Decimal 实例，然后可以从这个上下文继承精度和转换的取整参数。

```
import decimal

# Set up a context with limited precision
c = decimal.getcontext().copy()
c.prec = 3

# Create our constant
pi = c.create_decimal('3.1415')

# The constant value is rounded off
print 'PI      :', pi

# The result of using the constant uses the global context
print 'RESULT:', decimal.Decimal('2.01') * pi
```

例如，这样一来，应用就可以区别于用户数据精度而另外选择常量值精度。

```
$ python decimal_instance_context.py
```

```
PI      : 3.14
RESULT: 6.3114
```

线程

“全局”上下文实际上是线程本地上下文，所以完全可以使用不同的值分别配置各个线程。

```
import decimal
import threading
from Queue import PriorityQueue

class Multiplier(threading.Thread):
    def __init__(self, a, b, prec, q):
        self.a = a
        self.b = b
        self.prec = prec
        self.q = q
        threading.Thread.__init__(self)
    def run(self):
        c = decimal.getcontext().copy()
        c.prec = self.prec
        decimal.setcontext(c)
        self.q.put( (self.prec, a * b) )
        return
```



```

a = decimal.Decimal('3.14')
b = decimal.Decimal('1.234')
# A PriorityQueue will return values sorted by precision, no matter
# what order the threads finish.
q = PriorityQueue()
threads = [ Multiplier(a, b, i, q) for i in range(1, 6) ]
for t in threads:
    t.start()

for t in threads:
    t.join()

for i in range(5):
    prec, value = q.get()
    print prec, '\t', value

```

这个例子使用指定的值创建一个新的上下文，然后安装到各个线程中。

```
$ python decimal_thread_context.py
```

```

1      4
2      3.9
3      3.87
4      3.875
5      3.8748

```

参见：

`decimal` (<http://docs.python.org/library/decimal.html>) 这个模块的标准库文档。

Floating Point (http://en.wikipedia.org/wiki/Floating_point) 维基百科文章，有关浮点数表示和算术运算。

Floating Point Arithmetic: Issues and Limitations (<http://docs.python.org/tutorial/floatingpoint.html>) Python 教程中的一篇文章，介绍了浮点数数学表示问题。

5.2 fractions——有理数

作用：实现了一个类来处理有理数。

Python 版本：2.6 及以后版本

`Fraction` 类基于 `numbers` 模块中 `Rational` 定义的 API，实现了有理数的数值运算。

5.2.1 创建 Fraction 实例

与 `decimal` 模块类似，可以采用多种方式创建新值。一种简便的方式是由单独的分子和分母值来创建，如下所示。

```
import fractions

for n, d in [ (1, 2), (2, 4), (3, 6) ]:
    f = fractions.Fraction(n, d)
    print '%s/%s = %s' % (n, d, f)
```

计算新值时要保持最小公分母。

\$ python fractions_create_integers.py

```
1/2 = 1/2
2/4 = 1/2
3/6 = 1/2
```

创建 Fraction 的另一种方法是使用 <numerator>/<denominator> 字符串表示:

```
import fractions

for s in [ '1/2', '2/4', '3/6' ]:
    f = fractions.Fraction(s)
    print '%s = %s' % (s, f)
```

会解析这个字符串, 找出分子和分母值。

\$ python fractions_create_strings.py

```
1/2 = 1/2
2/4 = 1/2
3/6 = 1/2
```

字符串还可以使用更常用的小数或浮点数记法, 即用一个小数点分隔的一系列数字。

```
import fractions

for s in [ '0.5', '1.5', '2.0' ]:
    f = fractions.Fraction(s)
    print '%s = %s' % (s, f)
```

浮点数值表示的分子和分母值会自动计算。

\$ python fractions_create_strings_floats.py

```
0.5 = 1/2
1.5 = 3/2
2.0 = 2
```

还有一些类方法可以从有理数值的其他表示 (如 float 或 Decimal) 直接创建 Fraction 实例。

```
import fractions

for v in [ 0.1, 0.5, 1.5, 2.0 ]:
    print '%s = %s' % (v, fractions.Fraction.from_float(v))
```

不能精确表示的浮点数值可能会得到出乎意料的结果。

```
$ python fractions_from_float.py

0.1 = 3602879701896397/36028797018963968
0.5 = 1/2
1.5 = 3/2
2.0 = 2
```

使用值的 decimal 表示则会给出所期望的结果。

```
import decimal
import fractions

for v in [ decimal.Decimal('0.1'),
           decimal.Decimal('0.5'),
           decimal.Decimal('1.5'),
           decimal.Decimal('2.0'),
         ]:
    print '%s = %s' % (v, fractions.Fraction.from_decimal(v))
```

decimal 的内部实现不存在标准浮点数表示的精度错误。

```
$ python fractions_from_decimal.py

0.1 = 1/10
0.5 = 1/2
1.5 = 3/2
2.0 = 2
```

5.2.2 算术运算

一旦实例化分数，就可以在数学表达式中使用。

```
import fractions

f1 = fractions.Fraction(1, 2)
f2 = fractions.Fraction(3, 4)

print '%s + %s = %s' % (f1, f2, f1 + f2)
print '%s - %s = %s' % (f1, f2, f1 - f2)
print '%s * %s = %s' % (f1, f2, f1 * f2)
print '%s / %s = %s' % (f1, f2, f1 / f2)
```

分数运算支持所有标准操作符。

```
$ python fractions_arithmetic.py

1/2 + 3/4 = 5/4
1/2 - 3/4 = -1/4
1/2 * 3/4 = 3/8
1/2 / 3/4 = 2/3
```



5.2.3 近似值

Fraction 有一个有用的特性，它能够将一个浮点数转换为一个近似的有理数值。

```
import fractions
import math

print 'PI          =', math.pi

f_pi = fractions.Fraction(str(math.pi))
print 'No limit =', f_pi

for i in [ 1, 6, 11, 60, 70, 90, 100 ]:
    limited = f_pi.limit_denominator(i)
    print '{0:8} = {1}'.format(i, limited)
```

可以通过限制分母大小来控制这个分数的值。

```
$ python fractions_limit_denominator.py
```

```
PI          = 3.14159265359
No limit = 314159265359/1000000000000
 1 = 3
 6 = 19/6
11 = 22/7
60 = 179/57
70 = 201/64
90 = 267/85
100 = 311/99
```

参见：

fractions (<http://docs.python.org/library/fractions.html>) 这个模块的标准库文档。

decimal (5.1 节) decimal 模块提供了一个 API 来完成定点和浮点数的数学运算。

numbers (<http://docs.python.org/library/numbers.html>) 数值抽象基类。

5.3 random——伪随机数生成器

作用：实现了多种类型的伪随机数生成器。

Python 版本：1.4 及以后版本

random 模块基于 Mersenne Twister 算法提供了一个快速伪随机数生成器。原先开发这个生成器是为了向蒙特卡洛模拟生成输入，Mersenne Twister 算法会生成有一个大周期的近均匀分布的数，以适用于各种类型的应用。

5.3.1 生成随机数

random() 函数从所生成的序列返回下一个随机的浮点数值。返回的所有值都落在 $0 \leq n <$

1.0 区间内。

```
import random

for i in xrange(5):
    print '%04.3f' % random.random(),
print
```

重复运行这个程序会生成不同的数字序列。

```
$ python random_random.py
```

```
0.809 0.485 0.521 0.800 0.247
```

```
$ python random_random.py
```

```
0.614 0.551 0.705 0.479 0.659
```

要生成一个指定数值区间内的数，则使用 `uniform()`。

```
import random

for i in xrange(5):
    print '%04.3f' % random.uniform(1, 100),
print
```

传入最小值和最大值，`uniform()` 会使用公式 $\min + (\max - \min) * \text{random}()$ 来调整 `random()` 的返回值。

```
$ python random_uniform.py
```

```
78.558 96.734 74.521 52.386 98.499
```

5.3.2 指定种子

每次调用 `random()` 会生成不同的值，在一个非常大的周期之后数字才会重复。这对于生成惟一值或变化的值很有用，不过有些情况下可能需要提供相同的数据集，从而以不同的方式处理。对此，一种技术是使用一个程序来生成随机值，并保存这些随机值，以便通过一个单独的步骤另行处理。不过，这对于量很大的数据来说可能并不实用，所以 `random` 包含了一个 `seed()` 函数，用来初始化伪随机数生成器，使它能生成一个期望的值集。

```
import random

random.seed(1)

for i in xrange(5):
    print '%04.3f' % random.random(),
print
```

种子 (seed) 值会控制生成伪随机数所用公式产生的第一个值，由于公式是确定性的，改

变种子后也就设置要生成的整个序列。seed() 的参数可以是任意可散列对象。默认为使用一个平台特定的随机源（如果有的话）。否则，如果没有这样一个随机源，则会使用当前时间。

```
$ python random_seed.py

0.134 0.847 0.764 0.255 0.495

$ python random_seed.py

0.134 0.847 0.764 0.255 0.495.
```

5.3.3 保存状态

random() 使用的伪随机算法的内部状态可以保存，并用于控制后续各轮生成的随机数。继续生成随机数之前恢复前一个状态，这会减少由之前输入得到重复的值或值序列的可能性。getstate() 函数会返回一些数据，以后可以用 setstate() 利用这些数据重新初始化伪随机数生成器。

```
import random
import os
import cPickle as pickle

if os.path.exists('state.dat'):
    # Restore the previously saved state
    print 'Found state.dat, initializing random module'
    with open('state.dat', 'rb') as f:
        state = pickle.load(f)
        random.setstate(state)
else:
    # Use a well-known start state
    print 'No state.dat, seeding'
    random.seed(1)

# Produce random values
for i in xrange(3):
    print '%04.3f' % random.random(),
print
# Save state for next time
with open('state.dat', 'wb') as f:
    pickle.dump(random.getstate(), f)

# Produce more random values
print '\nAfter saving state:'
for i in xrange(3):
    print '%04.3f' % random.random(),
print
```



getstate() 返回的数据是一个实现细节，所以这个例子用 pickle 将数据保存到一个文件，不过可以把它当作一个黑盒。如果程序开始时这个文件存在，则加载原来的状态并继续。每次运行时都会在保存状态之前以及之后生成一些数，以展示恢复状态会导致生成器再次生成同样的值。

```
$ python random_state.py

No state.dat, seeding
0.134 0.847 0.764

After saving state:
0.255 0.495 0.449

$ python random_state.py

Found state.dat, initializing random module
0.255 0.495 0.449

After saving state:
0.652 0.789 0.094
```

5.3.4 随机整数

random() 将生成浮点数。可以把结果转换为整数，不过直接使用 randint() 生成整数会更方便。

```
import random

print '[1, 100]:',
for i in xrange(3):
    print random.randint(1, 100),

print '\n[-5, 5]:',
for i in xrange(3):
    print random.randint(-5, 5),
print
```

randint() 的参数是值的闭区间的两端。这些数可以是正数或负数，不过第一个值要小于第二个值。

```
$ python random_randint.py

[1, 100]: 91 77 67
[-5, 5]: -5 -3 3
```

randrange() 是从区间选择值的一种更一般的形式。

```
import random

for i in xrange(3):
    print random.randrange(0, 101, 5),
print
```

除了开始值 (start) 和结束值 (stop), `randrange()` 还支持一个步长 (step) 参数, 所以它完全等价于从 `range(start, stop, step)` 选择一个随机值。不过 `randrange` 更高效, 因为它并没有真正构造区间。

```
$ python random_randrange.py
```

```
50 10 60
```

5.3.5 选择随机元素

随机数生成器有一种常见用法, 即从一个枚举值序列中选择元素, 即使这些值并不是数字。`random` 包括一个 `choice()` 函数, 可以在一个序列中随机选择。下面这个例子模拟抛硬币 10000 次, 来统计多少次面朝上, 多少次面朝下。

```
import random
import itertools

outcomes = { 'heads':0,
              'tails':0,
              }
sides = outcomes.keys()

for i in range(10000):
    outcomes[ random.choice(sides) ] += 1

print 'Heads:', outcomes['heads']
print 'Tails:', outcomes['tails']
```

由于只允许两个结果, 所以不必使用数字然后再进行转换, 这里对 `choice()` 使用了单词 “heads” (表示面朝上) 和 “tails” (表示面朝下)。结果以表格形式存储在一个字典中, 使用结果名作为键。

```
$ python random_choice.py
```

```
Heads: 5038
Tails: 4962
```

5.3.6 排列

要模拟一个扑克牌游戏, 需要把一副牌混起来, 然后向玩家发牌, 同一张牌不能多次使用。使用 `choice()` 可能导致同一张牌被发出两次, 所以, 可以用 `shuffle()` 来洗牌, 然后在发各张牌时删除所发的牌。

```
import random
import itertools

FACE_CARDS = ('J', 'Q', 'K', 'A')
SUITS = ('H', 'D', 'C', 'S')
```

```

def new_deck():
    return list(itertools.product(
        itertools.chain(xrange(2, 11), FACE_CARDS),
        SUITS,
    ))
def show_deck(deck):
    p_deck = deck[:]
    while p_deck:
        row = p_deck[:13]
        p_deck = p_deck[13:]
        for j in row:
            print '%2s%s' % j,
        print

# Make a new deck, with the cards in order
deck = new_deck()
print 'Initial deck:'
show_deck(deck)

# Shuffle the deck to randomize the order
random.shuffle(deck)
print '\nShuffled deck:'
show_deck(deck)

# Deal 4 hands of 5 cards each
hands = [ [], [], [], [] ]

for i in xrange(5):
    for h in hands:
        h.append(deck.pop())

# Show the hands
print '\nHands:'
for n, h in enumerate(hands):
    print '%d:' % (n+1),
    for c in h:
        print '%2s%s' % c,
    print

# Show the remaining deck
print '\nRemaining deck:'
show_deck(deck)

```

这些扑克牌表示为元组，由面值和一个表示花色的字母组成。要创建已发出“一手牌”，可以一次向4个列表分别增加一张牌，然后从这副牌中将其删除，使这些牌不会再次发出。

```
$ python random_shuffle.py

Initial deck:
2H 2D 2C 2S 3H 3D 3C 3S 4H 4D 4C 4S 5H
5D 5C 5S 6H 6D 6C 6S 7H 7D 7C 7S 8H 8D
8C 8S 9H 9D 9C 9S 10H 10D 10C 10S JH JD JC
JS QH QD QC QS KH KD KC KS AH AD AC AS

Shuffled deck:
3C KH QH 6H JD AC 7S 5D 3S 10S 7H QC 2C
5C 7C 4H 6S 9D 10H 4D 2H 3D 7D 5S 10D 9H
2S 9C KC 5H 6C 8S 3H 10C JS 2D AH KD AD
4C QS 8D 8C JC 8H 4S JH QD 9S AS KS 6D

Hands:
1: 6D QD JC 4C 2D
2: KS JH 8C AD JS
3: AS 4S 8D KD 10C
4: 9S 8H QS AH 3H

Remaining deck:
3C KH QH 6H JD AC 7S 5D 3S 10S 7H QC 2C
5C 7C 4H 6S 9D 10H 4D 2H 3D 7D 5S 10D 9H
2S 9C KC 5H 6C 8S
```

5.3.7 采样

很多模拟需要从大量输入值中得到随机样本。`sample()` 函数可以生成无重复值的样本，且不会修改输入序列。下面的例子会打印系统字典中单词的一个随机样本。

```
import random

with open('/usr/share/dict/words', 'rt') as f:
    words = f.readlines()
words = [ w.rstrip() for w in words ]

for w in random.sample(words, 5):
    print w
```

生成结果集的算法会考虑输入的规模和所请求的样本，从而尽可能高效地生成结果。

```
$ python random_sample.py
```

```
pleasureman
consequency
docibility
youdendrift
Ituraean
```

```
$ python random_sample.py
```

```
jigamaree
readingdom
sporidium
pansylike
foraminiferan
```

5.3.8 多个并发生成器

除了模块级函数，random 还包括一个 Random 类来管理多个随机数生成器的内部状态。之前介绍的所有函数都可以作为 Random 实例的方法得到，而且各个实例可以单独初始化和使用，而不会与其他实例返回的值相互干扰。

```
import random
import time

print 'Default initializiation:\n'

r1 = random.Random()
r2 = random.Random()

for i in xrange(3):
    print '%04.3f %04.3f' % (r1.random(), r2.random())

print '\nSame seed:\n'

seed = time.time()
r1 = random.Random(seed)
r2 = random.Random(seed)

for i in xrange(3):
    print '%04.3f %04.3f' % (r1.random(), r2.random())
```

如果系统上设置了很好的内置随机值种子，不同实例会有惟一的初始状态。不过，如果没有一个好的平台随机值生成器，不同实例往往会用当前时间作为种子，因此会生成相同的值。

```
$ python random_random_class.py
```

```
Default initializiation:
```

```
0.370  0.303
0.437  0.142
0.323  0.088
```

```
Same seed:
```

```
0.684  0.684
```

```
0.060 0.060
0.977 0.977
```

为了确保生成器从随机周期的不同部分生成值，可以使用 `jumpahead()` 调整其中一个生成器的初始状态。

```
import random
import time

r1 = random.Random()
r2 = random.Random()

# Force r2 to a different part of the random period than r1.
r2.setstate(r1.getstate())
r2.jumpahead(1024)

for i in xrange(3):
    print '%04.3f %04.3f' % (r1.random(), r2.random())
```

`jumpahead()` 的参数应当是基于各生成器所需值个数的一个非负整数。生成器的内部状态根据这个输入值调整，但并不只是按给定的步数递增。

```
$ python random_jumpahead.py
0.858 0.093
0.510 0.707
0.444 0.556
```

5.3.9 SystemRandom

有些操作系统提供了一个随机数生成器，可以访问更多能够引入生成器的信息源。`random` 通过 `SystemRandom` 类提供了这个特性，这个类与 `Random` 的 API 相同，不过使用 `os.urandom()` 生成值，这构成了所有其他算法的基础。

```
import random
import time

print 'Default initialization:\n'

r1 = random.SystemRandom()
r2 = random.SystemRandom()

for i in xrange(3):
    print '%04.3f %04.3f' % (r1.random(), r2.random())

print '\nSame seed:\n'

seed = time.time()
r1 = random.SystemRandom(seed)
r2 = random.SystemRandom(seed)
```




```
for i in xrange(3):
    print '%04.3f %04.3f' % (r1.random(), r2.random())
```

SystemRandom 产生的序列是不可再生的，因为其随机性来自系统，而不是来自软件状态（实际上，seed() 和 setstate() 根本不起作用）。

```
$ python random_system_random.py
```

```
Default initialization:
```

```
0.551 0.873
0.643 0.975
0.106 0.268
Same seed:
```

```
0.211 0.985
0.101 0.852
0.887 0.344
```

5.3.10 非均匀分布

random() 生成的值为均匀分布，这对于很多用途来说非常有用，不过，另外一些分布可以更准确地对特定情况建模。random 模块还包含一些函数来生成这样一些分布的值。这里将列出这些分布，但是并不打算详细介绍，因为它们往往只在特定条件下使用，而且需要更复杂的例子来说明。

正态分布

正态分布（normal distribution）常用于非均匀的连续值，如梯度、高度、重量等等。正态分布产生的曲线有一个独特形状，所以被昵称为“钟形曲线”。random 包含两个函数可以生成正态分布的值，分别是 normalvariate() 和稍快一些的 gauss()。（正态分布也称为高斯分布。）

还有一个相关的函数 lognormvariate()，它可以生成对数呈正态分布的伪随机值。对数正态分布适用于多个不交互随机变量的积。

近似分布

三角分布用于小样本的近似分布。三角分布的“曲线”中，低点在已知的最小和最大值，在模式值处有一个高点，这要根据“最接近”的结果（由 triangular() 的模式参数反映）来估计。

指数分布

expovariate() 可以生成一个指数分布，这对于模拟到达或间隔时间值用于齐次泊松过程会很有用，如放射衰变速度或到达 Web 服务器的请求。

很多可观察的现象都适用帕累托分布或幂律分布，这个分布因 Chris Anderson 的“长尾效应”而普及。paretovariate() 函数对于模拟资源分配很有用（人的财富、音乐家的需求、对博客

的关注，等等)。

角分布

米塞斯分布或圆正态分布（由 `vonmisesvariate()` 生成）用于计算周期值的概率，如角度、日历日期和时间。

大小分布

`betavariate()` 生成 Beta 分布的值，常用于贝叶斯统计和应用，如任务持续时间建模。

`gammavariate()` 生成的伽玛分布用于对事物的大小建模，如等待时间、雨量和计算错误。

`weibullvariate()` 计算的韦伯分布用于故障分析、工业工程和天气预报。它描述了粒子或其他离散对象的大小分布。

参见：

`random` (<http://docs.python.org/library/random.html>) 这个模块的标准库文档。

Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 由 M. Matsumoto 和 T. Nishimura 撰写的一篇文章，发表在 ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January pp. 3–30 1998。

Mersenne Twister (http://en.wikipedia.org/wiki/Mersenne_twister) 维基百科文章，关于 Python 使用的伪随机数生成器算法。

Uniform distribution [[http://en.wikipedia.org/wiki/Uniform_distribution_\(continuous\)](http://en.wikipedia.org/wiki/Uniform_distribution_(continuous))] 维基百科文章，关于统计中的连续均匀分布。

5.4 math——数学函数

作用：提供函数完成特殊的数学运算。

Python 版本：1.4 及以后版本

`math` 模块实现了正常情况下内置平台 C 库中才有的很多 IEEE 函数，可以使用浮点值完成复杂的数学运算，包括对数和三角函数运算。

5.4.1 特殊常量

很多数学运算依赖于一些特殊的常量。`math` 包含有 π (pi) 和 e 的值。

```
import math
```

```
print 'π: %.30f' % math.pi
print 'e: %.30f' % math.e
```

这两个值的精度仅受平台的浮点数 C 库限制。

```
$ python math_constants.py
```

```
π: 3.141592653589793115997963468544
e: 2.718281828459045090795598298428
```

5.4.2 测试异常值

浮点数计算可能导致两种类型的异常值。第一种是 INF (无穷大), 如果用 double 存储一个浮点数值, 而它相对于一个有很大绝对值的值溢出时, 就会出现这个异常值。

```
import math

print '{:^3} {:6} {:6} {:6}'.format('e', 'x', 'x**2', 'isinf')
print '{:-^3} {:-^6} {:-^6} {:-^6}'.format('', '', '', '')

for e in range(0, 201, 20):
    x = 10.0 ** e
    y = x*x
    print '{:3d} {!s:6} {!s:6} {!s:6}'.format(e, x, y,
                                              math.isinf(y),
                                              )
```

这个例子中的指数变得足够大时, x 的平方无法再存放在一个 double 中, 这个值就会记录为无穷大。

```
$ python math_isinf.py
```

e	x	x**2	isinf
0	1.0	1.0	False
20	1e+20	1e+40	False
40	1e+40	1e+80	False
60	1e+60	1e+120	False
80	1e+80	1e+160	False
100	1e+100	1e+200	False
120	1e+120	1e+240	False
140	1e+140	1e+280	False
160	1e+160	inf	True
180	1e+180	inf	True
200	1e+200	inf	True

不过, 并不是所有浮点数溢出都会导致 INF 值。具体地, 用浮点数值计算一个指数时, 会生成 `OverflowError` 而不是保留 INF 结果。

```
x = 10.0 ** 200

print 'x      =', x
print 'x*x    =', x*x
try:
    print 'x**2 =', x**2
except OverflowError, err:
    print err
```

这种差异是由 C 和 Python 所用库中的实现差别造成的。

```
$ python math_overflow.py
```

```
x      = 1e+200
x*x    = inf
x**2   = (34, 'Result too large')
```

使用无穷大值的除法运算未定义。将一个数除以无穷大值的结果是 NaN（即不是一个数）。

```
import math
```

```
x = (10.0 ** 200) * (10.0 ** 200)
y = x/x
```

```
print 'x =', x
print 'isnan(x) =', math.isnan(x)
print 'y = x / x =', x/x
print 'y == nan =', y == float('nan')
print 'isnan(y) =', math.isnan(y)
```

NaN 不会等于任何值，甚至不等于其自身，所以要想检查 NaN，需要使用 `isnan()`。

```
$ python math_isnan.py
```

```
x = inf
isnan(x) = False
y = x / x = nan
y == nan = False
isnan(y) = True
```

5.4.3 转换为整数

`math` 模块包括 3 个函数用于将浮点数值转换为整数。这 3 个函数分别采用不同的方法，并适用于不同的场合。

最简单的是 `trunc()`，这会截断小数点后的数字，只留下构成这个值整数部分的有效数字。`floor()` 将其输入转换为不大于它的最大整数，`ceil()`（上限）会生成按顺序排在这个输入值之后的最小整数[⊖]。

```
import math
```

```
HEADINGS = ('i', 'int', 'trunc', 'floor', 'ceil')
print '{:^5}  {:^5}  {:^5}  {:^5}  {:^5}'.format(*HEADINGS)
print '{:-^5}  {:-^5}  {:-^5}  {:-^5}  {:-^5}'.format(
    '', '', '', '', ''
)

fmt = ' '.join(['{:5.1f}'] * 5)
```

⊖ 原文为 the largest integer（最大整数），有误。——译者注

```

TEST_VALUES = [ -1.5,
                 -0.8,
                 -0.5,
                 -0.2,
                 0,
                 0.2,
                 0.5,
                 0.8,
                 1,
                 ]
for i in TEST_VALUES:
    print fmt.format(i,
                     int(i),
                     math.trunc(i),
                     math.floor(i),
                     math.ceil(i))

```

`trunc()` 等价于直接转换为 `int`。

\$ python math_integers.py

i	int	trunk	floor	ceil
-1.5	-1.0	-1.0	-2.0	-1.0
-0.8	0.0	0.0	-1.0	-0.0
-0.5	0.0	0.0	-1.0	-0.0
-0.2	0.0	0.0	-1.0	-0.0
0.0	0.0	0.0	0.0	0.0
0.2	0.0	0.0	0.0	1.0
0.5	0.0	0.0	0.0	1.0
0.8	0.0	0.0	0.0	1.0
1.0	1.0	1.0	1.0	1.0

5.4.4 其他表示

`modf()` 取一个浮点数，并返回一个 `tuple`，其中包含这个输入值的小数和整数部分。

```
import math
```

```

for i in range(6):
    print '{} / 2 = {}'.format(i, math.modf(i/2.0))

```

返回值中的两个数都是浮点数。

\$ python math_modf.py

```

0/2 = (0.0, 0.0)
1/2 = (0.5, 0.0)
2/2 = (0.0, 1.0)

```

```
3/2 = (0.5, 1.0)
4/2 = (0.0, 2.0)
5/2 = (0.5, 2.0)
```

`frexp()` 返回一个浮点数的尾数和指数，可以用来对这个值创建一种更可移植的表示。

```
import math

print '{:^7} {:^7} {:^7}'.format('x', 'm', 'e')
print '{:-^7} {:-^7} {:-^7}'.format('', '', '')

for x in [ 0.1, 0.5, 4.0 ]:
    m, e = math.frexp(x)
    print '{:7.2f} {:7.2f} {:7d}'.format(x, m, e)
```

`frexp()` 使用公式 $x = m * 2^{**e}$ ，并返回值 `m` 和 `e`。

```
$ python math_frexp.py
```

x	m	e
0.10	0.80	-3
0.50	0.50	0
4.00	0.50	3

`ldexp()` 与 `frexp()` 正好相反。

```
import math

print '{:^7} {:^7} {:^7}'.format('m', 'e', 'x')
print '{:-^7} {:-^7} {:-^7}'.format('', '', '')

for m, e in [ (0.8, -3),
              (0.5, 0),
              (0.5, 3),
            ]:
    x = math.ldexp(m, e)
    print '{:7.2f} {:7d} {:7.2f}'.format(m, e, x)
```

使用与 `frexp()` 相同的公式，`ldexp()` 取尾数和指数值作为参数，将返回一个浮点数。

```
$ python math_ldexp.py
```

m	e	x
0.80	-3	0.10
0.50	0	0.50
0.50	3	4.00

5.4.5 正号和负号

一个数的绝对值就是不带正负号的本值。使用 `fabs()` 可以计算一个浮点数的绝对值。

```
import math

print math.fabs(-1.1)
print math.fabs(-0.0)
print math.fabs(0.0)
print math.fabs(1.1)
```

实际上, `float` 的绝对值表示为一个正值。

```
$ python math_fabs.py
```

```
1.1
0.0
0.0
1.1
```

要确定一个值的符号, 比如为一组值给定相同的符号或者要比较两个值, 可以使用 `copysign()` 来设置正确值的符号。

```
import math

HEADINGS = ('f', 's', '< 0', '> 0', '= 0')
print '{:^5}   {:^5}   {:^5}   {:^5}   {:^5}'.format(*HEADINGS)
print '{:-^5}   {:-^5}   {:-^5}   {:-^5}   {:-^5}'.format(
    '', '', '', '', ''
)

for f in [ -1.0,
           0.0,
           1.0,
           float('-inf'),
           float('inf'),
           float('-nan'),
           float('nan'),
           ]:
    s = int(math.copysign(1, f))
    print '{:5.1f}   {:5d}   {!s:5}   {!s:5}   {!s:5}'.format(
        f, s, f < 0, f > 0, f == 0,
    )
```

还需要一个类似 `copysign()` 的额外函数, 因为不能将 `NaN` 和 `-NaN` 与其他值直接比较。

```
$ python math_copysign.py
```

```

f      s      < 0      > 0      = 0
-----
-1.0    -1   True   False  False
```

0.0	1	False	False	True
1.0	1	False	True	False
-inf	-1	True	False	False
inf	1	False	True	False
nan	-1	False	False	False
nan	1	False	False	False

5.4.6 常用计算

在二进制浮点数内存中表示精确值很有难度。有些值无法准确地表示，而且一个值如果通过反复计算来处理，这样处理越频繁就越容易引入表示错误。math 包含一个函数来计算一系列浮点数的和，它使用一种高效的算法以尽量减少这种错误。

```
import math

values = [ 0.1 ] * 10

print 'Input values:', values

print 'sum()          : {:.20f}'.format(sum(values))

s = 0.0
for i in values:
    s += i
print 'for-loop      : {:.20f}'.format(s)

print 'math.fsum()   : {:.20f}'.format(math.fsum(values))
```

给定一个包含 10 个值的序列，每个值都等于 0.1，这个序列的总和期望值为 1.0。不过，由于 0.1 不能精确地表示为一个浮点值，所以会在总和中引入错误，除非用 fsum() 来计算。

```
$ python math_fsum.py
```

```
Input values: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
sum()          : 0.99999999999999988898
for-loop       : 0.99999999999999988898
math.fsum()    : 1.00000000000000000000
```

factorial() 常用于计算一系列对象的排列和组合数。一个正整数 n 的阶乘（表示为 $n!$ ）递归地定义为 $(n-1)! * n$ ，并在 $0! = 1$ 停止递归。

```
import math

for i in [ 0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.1 ]:
    try:
        print '{:2.0f}  {:6.0f}'.format(i, math.factorial(i))
    except ValueError, err:
        print 'Error computing factorial(%s):' % i, err
```


`factorial()` 只能处理整数，不过它确实接受 `float` 参数，只要这个参数可以转换为一个整数而不会丢值。

```
$ python math_factorial.py
```

```
0      1
1      1
2      2
3      6
4     24
5    120
```

```
Error computing factorial(6.1): factorial() only accepts integral
values
```

`gamma()` 类似于 `factorial()`，不过它可以处理实数，而且值会下移一个数（`gamma` 等于 $(n-1)!$ ）。

```
import math
```

```
for i in [ 0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 ]:
    try:
        print '{:2.1f}  {:6.2f}'.format(i, math.gamma(i))
    except ValueError, err:
        print 'Error computing gamma(%s):' % i, err
```

由于 0 会导致开始值为负，这是不允许的。

```
$ python math_gamma.py
```

```
Error computing gamma(0): math domain error
```

```
1.1    0.95
2.2    1.10
3.3    2.68
4.4   10.14
5.5   52.34
6.6  344.70
```

`lgamma()` 返回的结果是对输入值求 `gamma` 所得绝对值的自然对数。

```
import math
```

```
for i in [ 0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 ]:
    try:
        print '{:2.1f}  {:.20f}  {:.20f}'.format(
            i,
            math.lgamma(i),
            math.log(math.gamma(i)),
        )
    except ValueError, err:
        print 'Error computing lgamma(%s):' % i, err
```

使用 `lgamma()` 会比使用 `gamma()` 的结果单独计算对数更精确。

```
$ python math_lgamma.py
```

```
Error computing lgamma(0): math domain error
1.1 -0.04987244125984036103 -0.04987244125983997245
2.2 0.09694746679063825923 0.09694746679063866168
3.3 0.98709857789473387513 0.98709857789473409717
4.4 2.31610349142485727469 2.31610349142485727469
5.5 3.95781396761871651080 3.95781396761871606671
6.6 5.84268005527463252236 5.84268005527463252236
```

求模操作符 (%) 会计算一个除法表达式的余数 (例如, $5 \% 2 = 1$)。Python 语言内置的这个操作符可以很好地处理整数, 但是与很多其他浮点数运算类似, 中间计算可能导致表示问题, 进一步造成数据丢失。fmod() 可以为浮点值提供一个更精确的实现。

```
import math

print '{:^4}   {:^4}   {:^5}   {:^5}'.format('x', 'y', '%', 'fmod')
print '-----'

for x, y in [ (5, 2),
              (5, -2),
              (-5, 2),
            ]:
    print '{:4.1f}   {:4.1f}   {:5.2f}   {:5.2f}'.format(
        x,
        y,
        x % y,
        math.fmod(x, y),
    )
```

还有一点很可能经常产生混淆, fmod() 计算模所使用的算法与 % 使用的算法也有所不同, 所以结果的符号不同。

```
$ python math_fmod.py
```

x	y	%	fmod
5.0	2.0	1.00	1.00
5.0	-2.0	-1.00	1.00
-5.0	2.0	1.00	-1.00

5.4.7 指数和对数

指数生长曲线在经济学、物理学和其他科学中经常出现。Python 有一个内置的幂运算符 (**), 不过, 如果需要将一个可调用函数作为另一个函数的参数, 可能需要用到 `pow()`。

```

import math

for x, y in [
    # Typical uses
    (2, 3),
    (2.1, 3.2),

    # Always 1
    (1.0, 5),
    (2.0, 0),

    # Not-a-number
    (2, float('nan')),

    # Roots
    (9.0, 0.5),
    (27.0, 1.0/3),
]:
    print '{:5.1f} ** {:5.3f} = {:6.3f}'.format(x, y, math.pow(x, y))

```

1 的任何次幂总返回 1.0，同样的，任何值的指数为 0.0 时也总是返回 1.0。对于“不是一个数”值 nan，大多数运算都返回 nan。如果指数小于 1，pow() 会计算一个根。

```

$ python math_pow.py

2.0 ** 3.000 = 8.000
2.1 ** 3.200 = 10.742
1.0 ** 5.000 = 1.000
2.0 ** 0.000 = 1.000
2.0 ** nan = nan
9.0 ** 0.500 = 3.000
27.0 ** 0.333 = 3.000

```

由于平方根（指数为 1/2）使用非常频繁，所以有一个单独的函数来计算平方根。

```

import math

print math.sqrt(9.0)
print math.sqrt(3)
try:
    print math.sqrt(-1)
except ValueError, err:
    print 'Cannot compute sqrt(-1):', err

```

计算负数的平方根需要用到复数，这不在 math 的处理范围内。试图计算一个负值的平方根时，会导致一个 ValueError。

```

$ python math_sqrt.py

```

```

3.0
1.73205080757
Cannot compute sqrt(-1): math domain error

```

对数函数查找满足条件 $x = b ** y$ 的 y 。默认情况下, `log()` 计算自然对数 (底数为 e)。如果提供了第二个参数, 则使用这个参数值作为底数。

```
import math
```

```

print math.log(8)
print math.log(8, 2)
print math.log(0.5, 2)

```

x 小于 1 时, 求对数会生成负数结果。

```
$ python math_log.py
```

```

2.07944154168
3.0
-1.0

```

`log()` 有两个变型。给定浮点数表示和取整错误, `log(x, b)` 生成的计算值只有有限的精度 (特别是对于某些底数)。`log10()` 完成 `log(x, 10)` 计算, 但是会使用一种比 `log()` 更精确的算法。

```
import math
```

```

print '{:2} {:^12} {:^10} {:^20} {:8}'.format(
    'i', 'x', 'accurate', 'inaccurate', 'mismatch',
)
print '{:~^2} {:~^12} {:~^10} {:~^20} {:~^8}'.format(
    '', '', '', '', '',
)

```

```

for i in range(0, 10):
    x = math.pow(10, i)
    accurate = math.log10(x)
    inaccurate = math.log(x, 10)
    match = '' if int(inaccurate) == i else '*'
    print '{:2d} {:12.1f} {:10.8f} {:20.18f} {:^5}'.format(
        i, x, accurate, inaccurate, match,
    )

```

输出中末尾有 * 的行突出强调了不精确的值。

```
$ python math_log10.py
```

i	x	accurate	inaccurate	mismatch
0	1.0	0.00000000	0.000000000000000000	
1	10.0	1.00000000	1.000000000000000000	

30	0.52	0.52
45	0.79	0.79
60	1.05	1.05
90	1.57	1.57
180	3.14	3.14
270	4.71	4.71
360	6.28	6.28

要从弧度转换为度, 可以使用 `degrees()`。

```
import math

print '{:^8}   {:^8}   {:^8}'.format('Radians', 'Degrees', 'Expected')
print '{:-^8}   {:-^8}   {:-^8}'.format('', '', '')
for rad, expected in [ (0, 0),
                       (math.pi/6, 30),
                       (math.pi/4, 45),
                       (math.pi/3, 60),
                       (math.pi/2, 90),
                       (math.pi, 180),
                       (3 * math.pi / 2, 270),
                       (2 * math.pi, 360),
                       ]:
    print '{:8.2f}   {:8.2f}   {:8.2f}'.format(rad,
                                                math.degrees(rad),
                                                expected,
                                                )
```

具体转换公式为 $\text{deg} = \text{rad} * 180 / \pi$ 。

\$ python math_degrees.py

Radians	Degrees	Expected
0.00	0.00	0.00
0.52	30.00	30.00
0.79	45.00	45.00
1.05	60.00	60.00
1.57	90.00	90.00
3.14	180.00	180.00
4.71	270.00	270.00
6.28	360.00	360.00

5.4.9 三角函数

三角函数将三角形中的角与其边长相关联。在有周期性质的公式中经常出现三角函数, 如谐波或圆周运动, 或者处理角时也经常用到三角函数。标准库中所有三角函数的角参数都表示为弧度。

给定一个直角三角形中的角，其正弦是对边长度与斜边长度之比 ($\sin A = \text{opposite}/\text{hypotenuse}$)。余弦是邻边长度与斜边长度之比 ($\cos A = \text{adjacent}/\text{hypotenuse}$)。正切是对边与邻边之比 ($\tan A = \text{opposite}/\text{adjacent}$)。

```
import math

print 'Degrees  Radians  Sine      Cosine    Tangent'
print '-----  -'

fmt = ' '.join(['%7.2f'] * 5)

for deg in range(0, 361, 30):
    rad = math.radians(deg)
    if deg in (90, 270):
        t = float('inf')
    else:
        t = math.tan(rad)
    print fmt % (deg, rad, math.sin(rad), math.cos(rad), t)
```

正切也可以定义为这个角的正弦值与其余弦值之比，因为弧度 $\pi/2$ 和 $3\pi/2$ 的余弦是 0，所以相应的正切值为无穷大。

```
$ python math_trig.py
```

Degrees	Radians	Sine	Cosine	Tangent
-----	-	-	-	-
0.00	0.00	0.00	1.00	0.00
30.00	0.52	0.50	0.87	0.58
60.00	1.05	0.87	0.50	1.73
90.00	1.57	1.00	0.00	inf
120.00	2.09	0.87	-0.50	-1.73
150.00	2.62	0.50	-0.87	-0.58
180.00	3.14	0.00	-1.00	-0.00
210.00	3.67	-0.50	-0.87	0.58
240.00	4.19	-0.87	-0.50	1.73
270.00	4.71	-1.00	-0.00	inf
300.00	5.24	-0.87	0.50	-1.73
330.00	5.76	-0.50	0.87	-0.58
360.00	6.28	-0.00	1.00	-0.00

给定一个点 (x, y)，点 [(0, 0), (x, 0), (x, y)] 构成的三角形中斜边长度为 $(x^2 + y^2)^{1/2}$ ，可以用 `hypot()` 来计算。

```
import math

print '{:^7}  {:^7}  {:^10}'.format('X', 'Y', 'Hypotenuse')
print '{:-^7}  {:-^7}  {:-^10}'.format('', '', '')

for x, y in [ # simple points
```



```

(1, 1),
(-1, -1),
(math.sqrt(2), math.sqrt(2)),
(3, 4), # 3-4-5 triangle
# on the circle
(math.sqrt(2)/2, math.sqrt(2)/2), # pi/4 rads
(0.5, math.sqrt(3)/2), # pi/3 rads
]:
h = math.hypot(x, y)
print '{:7.2f} {:7.2f} {:7.2f}'.format(x, y, h)

```

对于圆上的点，总能得到斜边 == 1。

```
$ python math_hypot.py
```

X	Y	Hypotenuse
1.00	1.00	1.41
-1.00	-1.00	1.41
1.41	1.41	2.00
3.00	4.00	5.00
0.71	0.71	1.00
0.50	0.87	1.00

还可以用这个函数查看两个点之间的距离。

```

import math

print '{:^8} {:^8} {:^8} {:^8} {:^8}'.format(
    'X1', 'Y1', 'X2', 'Y2', 'Distance',
)
print '{:-^8} {:-^8} {:-^8} {:-^8} {:-^8}'.format(
    '', '', '', '', ''
)

for (x1, y1), (x2, y2) in [ ((5, 5), (6, 6)),
                           ((-6, -6), (-5, -5)),
                           ((0, 0), (3, 4)), # 3-4-5 triangle
                           ((-1, -1), (2, 3)), # 3-4-5 triangle
                           ]:
    x = x1 - x2
    y = y1 - y2
    h = math.hypot(x, y)
    print '{:8.2f} {:8.2f} {:8.2f} {:8.2f} {:8.2f}'.format(
        x1, y1, x2, y2, h,
    )

```

使用 x 值之差和 y 值之差将一个端点移至原点，然后将结果传入 hypot()。

```
$ python math_distance_2_points.py
```

X1	Y1	X2	Y2	Distance
5.00	5.00	6.00	6.00	1.41
-6.00	-6.00	-5.00	-5.00	1.41
0.00	0.00	3.00	4.00	5.00
-1.00	-1.00	2.00	3.00	5.00

math 还定义了反三角函数。

```
import math

for r in [ 0, 0.5, 1 ]:
    print 'arcsine(%1f)      = %5.2f' % (r, math.asin(r))
    print 'arccosine(%1f)    = %5.2f' % (r, math.acos(r))
    print 'arctangent(%1f)   = %5.2f' % (r, math.atan(r))
    print
```

1.57 大约等于 $\pi/2$, 或 90 度, 这个角的正弦为 1, 余弦为 0。

```
$ python math_inverse_trig.py
```

```
arcsine(0.0)      = 0.00
arccosine(0.0)    = 1.57
arctangent(0.0)   = 0.00

arcsine(0.5)      = 0.52
arccosine(0.5)    = 1.05
arctangent(0.5)   = 0.46

arcsine(1.0)      = 1.57
arccosine(1.0)    = 0.00
arctangent(1.0)   = 0.79
```

5.4.10 双曲函数

双曲函数经常出现在线性微分方程中, 处理电磁场、流体力学、狭义相对论和其他高级物理和数学问题时常会用到。

```
import math

print '{:^6}   {:^6}   {:^6}   {:^6}'.format(
    'X', 'sinh', 'cosh', 'tanh',
)
print '{:-^6}  {:^-6}  {:^-6}  {:^-6}'.format('', '', '', '')

fmt = '   '.join(['{:6.4f}'] * 4)

for i in range(0, 11, 2):
    x = i/10.0
    print fmt.format(x, math.sinh(x), math.cosh(x), math.tanh(x))
```

余弦和正弦函数构成一个圆，而双曲余弦和双曲正弦函数构成半个双曲线。

```
$ python math_hyperbolic.py
```

X	sinh	cosh	tanh
0.0000	0.0000	1.0000	0.0000
0.2000	0.2013	1.0201	0.1974
0.4000	0.4108	1.0811	0.3799
0.6000	0.6367	1.1855	0.5370
0.8000	0.8881	1.3374	0.6640
1.0000	1.1752	1.5431	0.7616

另外还提供了反双曲函数 `acosh()`、`asinh()` 和 `atanh()`。

5.4.11 特殊函数

统计学中经常用到高斯误差函数 (Gauss Error function)。

```
import math
```

```
print '{:^5} {:7}'.format('x', 'erf(x)')
```

```
print '{:-^5} {:^-7}'.format('', '')
```

```
for x in [ -3, -2, -1, -0.5, -0.25, 0, 0.25, 0.5, 1, 2, 3 ]:
```

```
    print '{:5.2f} {:7.4f}'.format(x, math.erf(x))
```

对于误差函数， $\text{erf}(-x) = -\text{erf}(x)$ 。

```
$ python math_erf.py
```

x	erf(x)
-3.00	-1.0000
-2.00	-0.9953
-1.00	-0.8427
-0.50	-0.5205
-0.25	-0.2763
0.00	0.0000
0.25	0.2763
0.50	0.5205
1.00	0.8427
2.00	0.9953
3.00	1.0000

补余误差函数是 $1-\text{erf}(x)$ 。

```
import math
```

```
print '{:^5} {:7}'.format('x', 'erfc(x)')
```



```
print '{:-^5}  {:^-7}'.format('', '')

for x in [ -3, -2, -1, -0.5, -0.25, 0, 0.25, 0.5, 1, 2, 3 ]:
    print '{:5.2f}  {:7.4f}'.format(x, math.erfc(x))
```

如果 x 值很小, `erfc()` 实现可以避免从 1 减时可能带来的精度误差。

```
$ python math_erfc.py
```

x	erfc(x)
-3.00	2.0000
-2.00	1.9953
-1.00	1.8427
-0.50	1.5205
-0.25	1.2763
0.00	1.0000
0.25	0.7237
0.50	0.4795
1.00	0.1573
2.00	0.0047
3.00	0.0000

参见:

`math` (<http://docs.python.org/library/math.html>) 这个模块的标准库文档。

IEEE floating-point arithmetic in Python(<http://www.johndcook.com/blog/2009/07/21/ieee-arithmetic-python/>) John Cook 撰写的博客文章, 介绍特殊值如何产生, 以及在 Python 中完成数学运算时如何处理特殊值。

SciPy (<http://scipy.org/>) Python 中实现科学和数学计算的开源库。



第⑥章

文件系统

Python 的标准库包括大量工具，可以处理文件系统中的文件、构造和解析文件名，还可以检查文件内容。

处理文件的第一步是确定要处理的文件的名字。Python 将文件名表示为简单的字符串，另外还提供了一些工具，用于由 `os.path` 中平台独立的标准组成部分构造文件名。用 `os` 中的 `listdir()` 可以列出一个目录中的内容，或者使用 `glob` 由一个模式建立文件名列表。

`glob` 使用的文件名模式匹配还可以通过 `fnmatch` 直接提供，从而可以在其他上下文中使用。

`dircache` 提供了一种高效的方式，能够扫描和处理文件系统中一个目录的内容，如果不能提前知道文件名，在这种情况下处理文件时 `dircache` 会很有用。

明确文件名之后，可以用 `os.stat()` 和 `stat` 中的常量检查其他特性，如权限或文件大小。

应用需要随机访问文件时，利用 `linecache` 可以很容易地按行号读取行。文件的内容在缓存中维护，所以要当心内存消耗。

有些情况下需要创建草稿文件来临时保存数据，或者将数据移动到一个永久位置之前需要用临时文件存储，此时 `tempfile` 就很有用。它提供了一些类，可以安全而稳妥地创建临时文件和目录。可以保证文件名是惟一的，其中包含随机的组成部分，因此不容易猜出。

程序通常需要把文件作为一个整体来处理，而不考虑其内容。`shutil` 模块包含了一些高级文件操作，如复制文件和目录，以及设置权限。

`filecmp` 模块通过查看文件和目录包含的字节来完成文件和目录比较，不过不涉及其格式的任何特殊知识。

内置的 `file` 类可以用于读写本地文件系统上可见的文件。不过，通过 `read()` 和 `write()` 接口访问大文件时，程序的性能可能会受影响，因为文件从磁盘移动到应用可见的内存时会涉及多次数据复制。使用 `mmap` 可以告诉操作系统使用其虚拟内存子系统，将文件的内容直接映射到程序可以访问的内存，从而避免在操作系统与 `file` 对象内部缓冲区之间进行复制。

如果文本数据中使用了非 ASCII 字符，通常会采用一种 Unicode 数据格式保存。由于标准 `file` 句柄假设文本文件的各个字节分别表示一个字符，所以读取使用多字节编码的 Unicode 文件需要额外的处理。`codecs` 模块会自动处理编码和解码，所以在很多情况下，完全可以使用一个非 ASCII 文件而无须任何其他修改。

如果测试代码依赖于从文件读写数据，对于这些测试代码，`StringIO` 提供了一个内存中流对象，它就像一个文件，不过不驻留在磁盘上。

6.1 os.path——平台独立的文件名管理

作用：解析、构建、测试以及处理文件名和路径。

Python 版本：1.4 及以后版本

使用 os.path 模块中包含的函数，很容易编写在多个平台上处理文件的代码。即使程序不打算在平台之间移植，也应当使用 os.path 来完成可靠的文件名解析。

6.1.1 解析路径

os.path 中的第一组函数可以用来将表示文件名的字符串解析为文件名的各个组成部分。有一点很重要，需要认识到这些函数并不要求路径真正存在；它们只处理字符串。

路径解析依赖于 os 中定义的一些变量：

- os.sep——路径各部分之间的分隔符（例如，“/”或“\”）。
- os.extsep——文件名与文件“扩展名”之间的分隔符（例如“.”）。
- os.pardir——路径中表示目录树上一级的部分（例如“..”）。
- os.curdir——路径中指示当前目录的部分（例如“.”）。

split() 函数将路径分解为两个单独的部分，并返回包含这些结果的一个 tuple。这个 tuple 的第二个元素是路径的最后一部分，第一个元素则是最后这个部分之前的所有内容。

```
import os.path

for path in [ '/one/two/three',
              '/one/two/three/',
              '//',
              '..',
              '' ]:
    print '%15s : %s' % (path, os.path.split(path))
```

输入参数以 os.sep 结尾时，路径的“最后一个元素”是一个空串。

```
$ python ospath_split.py

/one/two/three : ('/one/two', 'three')
/one/two/three/ : ('/one/two/three', '')
/ : ('/', '')
. : ('', '.')
: ('', '')
```

basename() 函数返回的值等价于 split() 值的第二部分。

```
import os.path

for path in [ '/one/two/three',
              '/one/two/three/',
              '//',
              '..',
              '' ]:
```



```
    '']:
    print '%15s : %s' % (path, os.path.basename(path))
```

整个路径会剥除到只剩下最后一个元素，不论这指示一个文件还是目录。如果路径以目录分隔符结尾（os.sep），则认为基本部分为空。

```
$ python ospath_basename.py
```

```
/one/two/three : three
/one/two/three/ :
/ :
. : .
:
```

dirname() 函数返回分解路径得到的第一部分。

```
import os.path
```

```
for path in [ '/one/two/three',
               '/one/two/three/',
               '//',
               '.',
               '']:
    print '%15s : %s' % (path, os.path.dirname(path))
```

将 basename() 的结果与 dirname() 结合，则可以得到原来的路径。

```
$ python ospath_dirname.py
```

```
/one/two/three : /one/two
/one/two/three/ : /one/two/three
/ : /
. :
:
```

splitext() 的作用类似于 split()，不过它会根据扩展名分隔符而不是目录分隔符来分解路径。

```
import os.path
```

```
for path in [ 'filename.txt',
               'filename',
               '/path/to/filename.txt',
               '//',
               '',
               'my-archive.tar.gz',
               'no-extension.',
               ]:
    print '%21s : ' % path, os.path.splitext(path)
```

查找扩展名时，只使用 `os.extsep` 的最后一次出现，所以如果一个文件名有多个扩展名，分解这个文件名时，部分扩展名会留在前缀上。

```
$ python ospath_splitext.py

filename.txt : ('filename', '.txt')
filename : ('filename', '')
/path/to/filename.txt : ('/path/to/filename', '.txt')
/ : ('/', '')
: ('', '')
my-archive.tar.gz : ('my-archive.tar', '.gz')
no-extension. : ('no-extension', '.')
```

`commonprefix()` 取一个路径列表作为参数，将返回一个字符串，表示所有路径中都出现的公共前缀。这个值可能表示一个根本不存在的路径，而且并不考虑路径分隔符，所以这个前缀可能并不落在一个分隔符边界上。

```
import os.path

paths = ['/one/two/three/four',
         '/one/two/threefold',
         '/one/two/three/',
        ]

for path in paths:
    print 'PATH:', path

print
print 'PREFIX:', os.path.commonprefix(paths)
```

在这个例子中，公共前缀字符串是 `/one/two/three`，尽管其中一个路径并不包括一个名为 `three` 的目录。

```
$ python ospath_commonprefix.py

PATH: /one/two/three/four
PATH: /one/two/threefold
PATH: /one/two/three/

PREFIX: /one/two/three
```

6.1.2 建立路径

除了分解现有的路径，还经常需要从其他字符串建立路径。要将多个路径组成部分结合为一个值，可以使用 `join()`。

```
import os.path

for parts in [ ('one', 'two', 'three'),
               ('/', 'one', 'two', 'three'),
```



```

        ('/one', '/two', '/three'),
    ]:
    print parts, ': ', os.path.join(*parts)

```

如果要连接的某个参数以 `os.sep` 开头，前面的所有参数都会丢弃，这个新参数会成为返回值的开始部分。

```
$ python ospath_join.py
```

```

('one', 'two', 'three') : one/two/three
('/', 'one', 'two', 'three') : /one/two/three
('/one', '/two', '/three') : /three

```

还可以处理包含“可变”部分的路径，这些“可变”部分可以自动扩展。例如，`expanduser()` 可以将波浪线(~) 字符转换为用户主目录名。

```

import os.path

for user in ['', 'dhellmann', 'postgresql']:
    lookup = '~' + user
    print '%12s : %s' % (lookup, os.path.expanduser(lookup))

```

如果用户的主目录无法找到，则字符串不做任何改动直接返回，如下面这个例子中的 `~postgresql`。

```
$ python ospath_expanduser.py

~ : /Users/dhellmann
~dhellmann : /Users/dhellmann
~postgresql : ~postgresql

```

`expandvars()` 更为通用，它会扩展路径中出现的所有 shell 环境变量。

```

import os.path
import os

os.environ['MYVAR'] = 'VALUE'

print os.path.expandvars('/path/to/$MYVAR')

```

这里不会做任何验证来确保变量值能够得到真正已经存在的文件名。

```
$ python ospath_expandvars.py

/path/to/VALUE

```

6.1.3 规范化路径

使用 `join()` 或利用嵌入变量由单独的字符串组合路径时，得到的路径最后可能会有多余的

分隔符或相对路径部分。使用 `normpath()` 可以清除这些内容。

```
import os.path

for path in [ 'one//two//three',
              'one../two../three',
              'one../alt/two/three',
              ]:
    print '%20s : %s' % (path, os.path.normpath(path))
```

这里会计算并“压缩”由 `os.curdir` 和 `os.pardir` 构成的路径段。

```
$ python ospath_normpath.py
```

```
one//two//three : one/two/three
one../two../three : one/two/three
one../alt/two/three : alt/two/three
```

要把一个相对路径转换为一个绝对文件名，可以使用 `abspath()`。

```
import os
import os.path

os.chdir('/tmp')

for path in [ '..',
              '...',
              './one/two/three',
              '../one/two/three',
              ]:
    print '%17s : "%s"' % (path, os.path.abspath(path))
```

结果是一个从文件系统树最顶层开始的完整的路径。

```
$ python ospath_abspath.py
```

```
      . : "/private/tmp"
      .. : "/private"
./one/two/three : "/private/tmp/one/two/three"
../one/two/three : "/private/one/two/three"
```

6.1.4 文件时间

除了处理路径，`os.path` 还包括一些用来获取文件属性的函数，类似于 `os.stat()` 返回的结果。

```
import os.path
import time

print 'File      :', __file__
print 'Access time :', time.ctime(os.path.getatime(__file__))
print 'Modified time:', time.ctime(os.path.getmtime(__file__))
print 'Change time :', time.ctime(os.path.getctime(__file__))
```

```
print 'Size      :', os.path.getsize(__file__)
```

os.path.getatime() 返回访问时间, os.path.getmtime() 返回修改时间, os.path.getctime() 返回创建时间。os.path.getsize() 返回文件中的数据量, 以字节为单位表示。

```
$ python ospath_properties.py
```

```
File           : ospath_properties.py
Access time    : Sat Nov 27 12:19:50 2010
Modified time  : Sun Nov 14 09:40:36 2010
Change time    : Tue Nov 16 08:07:32 2010
Size           : 495
```

6.1.5 测试文件

程序遇到一个路径名时, 通常要知道这个路径是指示一个文件、目录还是一个符号链接(symlink), 另外还要知道它是否确实存在。os.path 包含了一些函数来测试所有这些条件。

```
import os.path
```

```
FILENAMES = [ __file__,
               os.path.dirname(__file__),
               '/',
               './broken_link',
               ]
```

```
for file in FILENAMES:
    print 'File      :', file
    print 'Absolute   :', os.path.isabs(file)
    print 'Is File?    :', os.path.isfile(file)
    print 'Is Dir?     :', os.path.isdir(file)
    print 'Is Link?    :', os.path.islink(file)
    print 'Mountpoint? :', os.path.ismount(file)
    print 'Exists?     :', os.path.exists(file)
    print 'Link Exists?:', os.path.lexists(file)
    print
```

所有这些测试函数都返回布尔值。

```
$ ln -s /does/not/exist broken_link
$ python ospath_tests.py
```

```
File           : ospath_tests.py
Absolute       : False
Is File?      : True
Is Dir?       : False
Is Link?      : False
Mountpoint?   : False
Exists?       : True
Link Exists?: True
```



```
File      :  
Absolute  : False  
Is File?  : False  
Is Dir?   : False  
Is Link?  : False  
Mountpoint? : False  
Exists?   : False  
Link Exists?: False  
  
File      : /  
Absolute  : True  
Is File?  : False  
Is Dir?   : True  
Is Link?  : False  
Mountpoint? : True  
Exists?   : True  
Link Exists?: True  
  
File      : ./broken_link  
Absolute  : False  
Is File?  : False  
Is Dir?   : False  
Is Link?  : True  
Mountpoint? : False  
Exists?   : False  
Link Exists?: True
```

6.1.6 遍历一个目录树

`os.path.walk()` 会遍历一个树中的所有目录，并调用所提供的函数，将目录名和该目录中各内容的名字作为参数传入该函数。

```
import os  
import os.path  
import pprint  
  
def visit(arg, dirname, names):  
    print dirname, arg  
    for name in names:  
        subname = os.path.join(dirname, name)  
        if os.path.isdir(subname):  
            print ' %s/' % name  
        else:  
            print ' %s' % name  
    print  
  
if not os.path.exists('example'):
```



```

    os.mkdir('example')
if not os.path.exists('example/one'):
    os.mkdir('example/one')

with open('example/one/file.txt', 'wt') as f:
    f.write('contents')
with open('example/two.txt', 'wt') as f:
    f.write('contents')

os.path.walk('example', visit, '(User data)')

```

这个例子会生成一个递归的目录列表，这里忽略了 .svn 目录。

```
$ python ospath_walk.py
```

```

example (User data)
  one/
  two.txt

example/one (User data)
  file.txt

```

参见：

os.path (<http://docs.python.org/lib/module-os.path.html>) 这个模块的标准库文档。

os (17.3 节) os 模块是 os.path 的父模块。

time (4.1 节) time 模块包含有一些函数，可以在 os.path 中时间属性函数所用的表示与易读的字符串表示之间完成转换。

6.2 glob——文件名模式匹配

作用：使用 UNIX shell 规则查找与一个模式匹配的文件名。

Python 版本：1.4 及以后版本

尽管 glob API 很小，但这个模块的功能却很强大。只要程序需要查找文件系统中名字与某个模式匹配的一组文件，就可以使用这个模块。要创建一个文件名列表，要求其中各个文件名都有某个特定的扩展名、前缀或者中间都有某个共同的字符串，就可以使用 glob 而不用编写定制代码来扫描目录内容。

glob 的模式规则与 re 模块使用的正则表达式并不相同。实际上，glob 的模式遵循标准 UNIX 路径扩展规则。只使用几个特殊字符来实现两个不同的通配符和字符区间。模式规则要应用于文件名中的段（在路径分隔符 / 处截止）。模式中的路径可以是相对路径或绝对路径。shell 变量名和波浪线 (~) 都不会扩展。

6.2.1 示例数据

本节中的例子假设当前工作目录中有以下测试文件。

```
$ python glob_maketestdata.py
```

```
dir
dir/file.txt
dir/file1.txt
dir/file2.txt
dir/filea.txt
dir/fileb.txt
dir/subdir
dir/subdir/subfile.txt
```

如果这些文件不存在，那么在运行以下例子之前，请使用示例代码中的 `glob_maketestdata.py` 创建这些文件。

6.2.2 通配符

星号(*)匹配一个文件名段中的0个或多个字符。例如，`dir/*`。

```
import glob
for name in glob.glob('dir/*'):
    print name
```

这个模式会匹配目录“dir”中的所有路径名（文件或目录），但不会进一步递归搜索到子目录。

```
$ python glob_asterisk.py
```

```
dir/file.txt
dir/file1.txt
dir/file2.txt
dir/filea.txt
dir/fileb.txt
dir/subdir
```

要列出子目录中的文件，必须把子目录包含在模式中。

```
import glob

print 'Named explicitly:'
for name in glob.glob('dir/subdir/*'):
    print '\t', name

print 'Named with wildcard:'
for name in glob.glob('dir/**/*.'):
    print '\t', name
```

前面显示的第一种情况显式列出了子目录名，第二种情况则依赖一个通配符查找目录。

```
$ python glob_subdir.py
```

```
Named explicitly:
    dir/subdir/subfile.txt
Named with wildcard:
    dir/subdir/subfile.txt
```

在这里，两种做法的结果是一样的。如果还有另一个子目录，通配符则会匹配这两个子目录，并包含这两个子目录中的文件名。

6.2.3 单字符通配符

问号(?)也是一个通配符，它会匹配文件名中该位置的单个字符。

```
import glob

for name in glob.glob('dir/file?.txt'):
    print name
```

前面的例子会匹配以 file 开头，然后是另外一个任意字符，最后以 .txt 结尾的所有文件名。

```
$ python glob_question.py
```

```
dir/file1.txt
dir/file2.txt
dir/filea.txt
dir/fileb.txt
```

6.2.4 字符区间

如果使用字符区间([a-z])而不是问号，可以匹配多个字符中的一个字符。下面这个例子会查找扩展名前名字中有一个数字的所有文件。

```
import glob
for name in glob.glob('dir/*[0-9].*'):
    print name
```

字符区间 [0-9] 会匹配所有单个数字。区间根据各字母/数字的字符码排序，短横线指示连续字符组成的一个不间断区间。这个区间值也可以写作 [0123456789]。

```
$ python glob_charrange.py
```

```
dir/file1.txt
dir/file2.txt
```

参见：

glob (<http://docs.python.org/library/glob.html>) 这个模块的标准库文档。

Pattern Matching Notation(http://www.opengroup.org/onlinepubs/000095399/utilities/xcu_chap02.html#tag_02_13) Open Group 的 shell 命令语言规范中对文件名模式匹配的解释。

fnmatch (6.9 节) 文件名匹配实现。

6.3 linecache——高效读取文本文件

作用：从文件或导入的 Python 模块获取文本行，维护一个结果缓存，从而可以更高效地从相同文件读取多行文本。

Python 版本：1.4 及以后版本

处理 Python 源文件时，linecache 模块会在 Python 标准库的其他部分中用到。缓存实现将在内存中保存文件的内容（解析为单独的行）。API 通过索引一个 list 返回所请求的行，与反复地读取文件并解析文本来查找所需文本行相比，这样可以节省时间。这个方法在查找同一文件中的多行时尤其有用，比如为一个错误报告生成一个跟踪记录（traceback）。

6.3.1 测试数据

后面将使用由一个 Lorem Ipsum 生成器生成的以下文本作为示例输入。

```
import os
import tempfile

lorem = '''Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vivamus eget elit. In posuere mi non
risus. Mauris id quam posuere lectus sollicitudin
varius. Praesent at mi. Nunc eu velit. Sed augue massa,
fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur
eros pede, egestas at, ultricies ac, apellentesque eu,
tellus.

Sed sed odio sed mi luctus mollis. Integer et nulla ac augue
convallis accumsan. Ut felis. Donec lectus sapien, elementum
nec, condimentum ac, interdum non, tellus. Aenean viverra,
mauris vehicula semper porttitor, ipsum odio consectetur
lorem, ac imperdiet eros odio a sapien. Nulla mauris tellus,
aliquam non, egestas a, nonummy et, erat. Vivamus sagittis
porttitor eros.'''

def make_tempfile():
    fd, temp_file_name = tempfile.mkstemp()
    os.close(fd)
    f = open(temp_file_name, 'wt')
    try:
        f.write(lorem)
    finally:
        f.close()
    return temp_file_name

def cleanup(filename):
    os.unlink(filename)
```


6.3.2 读取特定行

linecache 模块读取的文件行号从 1 开始，不过通常列表的数组索引会从 0 开始。

```
import linecache
from linecache_data import *

filename = make_tempfile()

# Pick out the same line from source and cache.
# (Notice that linecache counts from 1)
print 'SOURCE:'
print '%r' % lorem.split('\n')[4]
print
print 'CACHE:'
print '%r' % linecache.getline(filename, 5)

cleanup(filename)
```

返回的各行包括一个末尾的换行符。

```
$ python linecache_getline.py
```

```
SOURCE:
'fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur'

CACHE:
'fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur\n'
```

6.3.3 处理空行

返回值通常在行末尾都包括一个换行符，所以如果文本行为空，那么返回值就是一个换行符。

```
import linecache
from linecache_data import *

filename = make_tempfile()

# Blank lines include the newline
print 'BLANK : %r' % linecache.getline(filename, 8)

cleanup(filename)
```

输入文件的第 8 行没有包含任何文本。

```
$ python linecache_empty_line.py
```

```
BLANK : '\n'
```



6.3.4 错误处理

如果所请求的行号超出了文件中合法行号的范围，`getline()` 会返回一个空串。

```
import linecache
from linecache_data import *

filename = make_tempfile()

# The cache always returns a string, and uses
# an empty string to indicate a line which does
# not exist.
not_there = linecache.getline(filename, 500)
print 'NOT THERE: %r includes %d characters' % \
    (not_there, len(not_there))

cleanup(filename)
```

输入文件只有 12 行，所以请求第 500 行就像是试图越过文件末尾继续读文件。

```
$ python linecache_out_of_range.py
```

```
NOT THERE: '' includes 0 characters
```

读取一个不存在的文件时，也采用同样的方式处理。

```
import linecache

# Errors are even hidden if linecache cannot find the file
no_such_file = linecache.getline('this_file_does_not_exist.txt', 1)
print 'NO FILE: %r' % no_such_file
```

调用者试图读取数据时，这个模块不会产生异常。

```
$ python linecache_missing_file.py
```

```
NO FILE: ''
```

6.3.5 读取 Python 源文件

由于 `linecache` 在生成 `traceback` 跟踪记录时使用相当频繁，其关键特性之一就是能够通过指定模块的基名在导入路径中查找 Python 源模块。

```
import linecache
import os

# Look for the linecache module, using
# the built in sys.path search.
module_line = linecache.getline('linecache.py', 3)
print 'MODULE:'
print repr(module_line)
```

```
# Look at the linecache module source directly.
file_src = linecache.__file__
if file_src.endswith('.pyc'):
    file_src = file_src[:-1]
print '\nFILE:'
with open(file_src, 'r') as f:
    file_line = f.readlines()[2]
print repr(file_line)
```

如果 linecache 中的缓存填充代码在当前目录中无法找到指定名的文件，它会在 sys.path 中搜索指定名的模块。这个例子要查找 linecache.py。由于当前目录中没有这个文件副本，所以会找到标准库中的相应文件。

```
$ python linecache_path_search.py
```

```
MODULE:
'This is intended to read lines from modules imported -- hence if a
filename\n'

FILE:
'This is intended to read lines from modules imported -- hence if a
filename\n'
```

参见：

linecache (<http://docs.python.org/library/linecache.html>) 这个模块的标准库文档。
<http://www.ipsum.com/> Lorem Ipsum 生成器。

6.4 tempfile——临时文件系统对象

作用：创建临时文件系统对象。

Python 版本：1.4 及以后版本

要想安全地创建具有惟一名称的临时文件，以防止被试图破坏应用或窃取数据的人猜出，这并不容易。tempfile 模块提供了多个函数来安全地创建临时文件系统资源。TemporaryFile() 打开并返回一个未命名的文件，NamedTemporaryFile() 打开并返回一个命名文件，mkdtemp() 会创建一个临时目录，并返回其目录名。

6.4.1 临时文件

如果应用需要临时文件来存储数据，而不需要与其他程序共享这些文件，就应当使用 TemporaryFile() 函数创建文件。这个函数会创建一个文件，而且如果平台支持，它会立即断开文件链接。这样一来，其他程序就不可能找到或打开这个文件了，因为文件系统表中根本没有这个文件的引用。对于 TemporaryFile() 创建的文件，不论通过调用 close() 还是结合使用上下

文管理器 API 和 with 语句来关闭文件，文件都会在关闭时自动删除。

```
import os
import tempfile

print 'Building a filename with PID:'
filename = '/tmp/guess_my_name.%s.txt' % os.getpid()
temp = open(filename, 'w+b')
try:
    print 'temp:'
    print ' ', temp
    print 'temp.name:'
    print ' ', temp.name
finally:
    temp.close()
    # Clean up the temporary file yourself
    os.remove(filename)

print
print 'TemporaryFile:'
temp = tempfile.TemporaryFile()
try:
    print 'temp:'
    print ' ', temp
    print 'temp.name:'
    print ' ', temp.name
finally:
    # Automatically cleans up the file
    temp.close()
```

这个例子展示了采用不同方法创建临时文件的差别，一种做法是使用一个通用模式来构造临时文件的文件名，另一种做法是使用 TemporaryFile() 函数。TemporaryFile() 返回的文件没有文件名。

```
$ python tempfile_TemporaryFile.py
```

```
Building a filename with PID:
```

```
temp:
```

```
<open file '/tmp/guess_my_name.1074.txt', mode 'w+b' at
0x100d881e0>
```

```
temp.name:
```

```
/tmp/guess_my_name.1074.txt
```

```
TemporaryFile:
```

```
temp:
```

```
<open file '<fdopen>', mode 'w+b' at 0x100d88780>
```

```
temp.name:
```

```
<fdopen>
```



默认地，文件句柄采用模式 'w+b' 创建，使之在所有平台上都表现一致，而且调用者可以读写这个文件。

```
import os
import tempfile

with tempfile.TemporaryFile() as temp:
    temp.write('Some data')
    temp.seek(0)

    print temp.read()
```

写文件之后，必须使用 seek() “回转” 文件句柄，从而能够由文件读回数据。

```
$ python tempfile_TemporaryFile_binary.py
```

```
Some data
```

要以文本模式打开文件，创建文件时要设置模式为 'w+t'。

```
import tempfile

with tempfile.TemporaryFile(mode='w+t') as f:
    f.writelines(['first\n', 'second\n'])
    f.seek(0)

    for line in f:
        print line.rstrip()
```

这个文件句柄将数据处理为文本。

```
$ python tempfile_TemporaryFile_text.py
```

```
first
second
```

6.4.2 命名文件

很多情况下都需要有一个命名的临时文件。对于跨多个进程甚至主机的应用来说，为文件命名是在应用不同部分之间传递文件的最简单的方法。NamedTemporaryFile() 函数会创建一个文件，但不会断开其链接，所以会保留其文件名（用 name 属性访问）。

```
import os
import tempfile

with tempfile.NamedTemporaryFile() as temp:
    print 'temp:'
    print ' ', temp
    print 'temp.name:'
    print ' ', temp.name
```

```
print 'Exists after close:', os.path.exists(temp.name)
```

句柄关闭后文件会被删除。

```
$ python tempfile_NamedTemporaryFile.py
```

```
temp:
  <open file '<fdopen>', mode 'w+b' at 0x100d881e0>
temp.name:
  /var/folders/9R/9Rlt+tR02Raxzk+F71Q50U+++Uw/-Tmp-/tmp926BkT
Exists after close: False
```

6.4.3 临时目录

需要多个临时文件时，可能更方便的做法是用 `mkdtemp()` 创建一个临时目录，并打开该目录中的所有文件。

```
import os
import tempfile

directory_name = tempfile.mkdtemp()
print directory_name
# Clean up the directory
os.removedirs(directory_name)
```

由于这个目录事实上并不是“打开的”，不再需要它时必须显式地将其删除。

```
$ python tempfile_mkdtemp.py
```

```
/var/folders/9R/9Rlt+tR02Raxzk+F71Q50U+++Uw/-Tmp-/tmpA7DKtP
```

6.4.4 预测名

虽然没有严格匿名的临时文件那么安全，但有时也需要在名字中包含一个可预测的部分，从而能够查找和检查文件来进行调试。目前为止介绍的所有函数都取 3 个参数，可以在某种程度上控制文件。文件名使用以下公式生成。

```
dir + prefix + random + suffix
```

除了 `random` 外，所有其他值都可以作为参数传递给 `TemporaryFile()`、`NamedTemporaryFile()` 和 `mkdtemp()`。例如：

```
import tempfile

with tempfile.NamedTemporaryFile(
    suffix='_suffix', prefix='prefix_', dir='/tmp',
) as temp:
    print 'temp:'
    print ' ', temp
    print 'temp.name:'
    print ' ', temp.name
```

前缀 (prefix) 和后缀 (suffix) 参数与一个随机的字符串结合起来生成文件名, dir 参数保持不变, 用作新文件的位置。

```
$ python tempfile_NamedTemporaryFile_args.py

temp:
    <open file '<fdopen>', mode 'w+b' at 0x100d881e0>
temp.name:
    /tmp/prefix_kjvHYS_suffix
```

6.4.5 临时文件位置

如果没有使用 dir 参数指定明确的目标位置, 临时文件使用的路径会根据当前平台和设置而有所不同。tempfile 模块包含两个函数来查询运行时使用的设置。

```
import tempfile

print 'gettempdir():', tempfile.gettempdir()
print 'gettempprefix():', tempfile.gettempprefix()
```

gettempdir() 返回包含所有临时文件的默认目录, gettempprefix() 返回新文件和目录名的字符串前缀。

```
$ python tempfile_settings.py

gettempdir(): /var/folders/9R/9R1t+tR02Raxzk+F71Q50U+++Uw/-Tmp-
gettempprefix(): tmp
```

gettempdir() 返回的值根据一个简单算法来设置, 它会查找 5 个位置, 寻找允许当前进程创建文件的第一个位置。搜索列表如下:

1. 环境变量 TMPDIR。
2. 环境变量 TEMP。
3. 环境变量 TMP。
4. 作为“后路”的位置, 取决于具体平台。(RiscOS 使用 Wimp\$ScrapDir。Windows 使用 C:\TEMP、C:\TMP、\TEMP 或 \TMP 中第一个可用的位置。其他平台使用 /tmp、/var/tmp 或 /usr/tmp。)
5. 如果找不到其他目录, 则使用当前工作目录。

```
import tempfile

tempfile.tempdir = '/I/changed/this/path'
print 'gettempdir():', tempfile.gettempdir()
```

如果程序需要对所有临时文件使用一个全局位置, 但不使用以上任何环境变量, 则应当直接设置 tempfile.tempdir, 为该变量赋一个值。

```
$ python tempfile_tempdir.py
```

```
gettempdir(): /I/changed/this/path
```

参见:

tempfile (<http://docs.python.org/lib/module-tempfile.html>) 这个模块的标准库文档。

6.5 shutil——高级文件操作

作用: 高级文件操作。

Python 版本: 1.4 及以后版本

shutil 模块包括一些高级文件操作, 如复制和设置权限。

6.5.1 复制文件

copyfile() 将源的内容复制到目标, 如果没有权限写目标文件则产生 IOError。

```
from shutil import *
from glob import glob

print 'BEFORE:', glob('shutil_copyfile.*')
copyfile('shutil_copyfile.py', 'shutil_copyfile.py.copy')
print 'AFTER:', glob('shutil_copyfile.*')
```

由于这个函数会打开输入文件进行读取, 而不论其类型, 所以某些特殊文件 (如 UNIX 设备节点) 不能用 copyfile() 复制为新的特殊文件。

```
$ python shutil_copyfile.py
```

```
BEFORE: ['shutil_copyfile.py']
AFTER: ['shutil_copyfile.py', 'shutil_copyfile.py.copy']
```

copyfile() 的实现使用了底层函数 copyfileobj()。copyfile() 的参数是文件名, 但 copyfileobj() 的参数是打开的文件句柄。还可以有第三个参数 (可选): 用于读入块的一个缓冲区长度。

```
from shutil import *
import os
from StringIO import StringIO
import sys

class VerboseStringIO(StringIO):
    def read(self, n=-1):
        next = StringIO.read(self, n)
        print 'read(%d) bytes' % n
        return next
```

```
lorem_ipsum = '''Lorem ipsum dolor sit amet, consectetur adipiscing
```



```
elit. Vestibulum aliquam mollis dolor. Donec vulputate nunc ut diam.
Ut rutrum mi vel sem. Vestibulum ante ipsum.'''
```

```
print 'Default:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output)
```

```
print
```

```
print 'All at once:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output, -1)
```

```
print
```

```
print 'Blocks of 256:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output, 256)
```

默认行为是使用大数据块读取。使用 -1 会一次读入所有输入，或者使用其他正数可以设置特定的块大小。下面这个例子将使用多个不同的块大小来显示效果。

```
$ python shutil_copyfileobj.py
```

```
Default:
read(16384) bytes
read(16384) bytes
```

```
All at once:
read(-1) bytes
read(-1) bytes
```

```
Blocks of 256:
read(256) bytes
read(256) bytes
```

类似于 UNIX 命令行工具 cp，copy() 函数会用同样的方式解释输出名。如果指定的目标指示一个目录而不是一个文件，会使用源文件的基名在该目录中创建一个新文件。

```
from shutil import *
import os

os.mkdir('example')
print 'BEFORE:', os.listdir('example')
copy('shutil_copy.py', 'example')
print 'AFTER:', os.listdir('example')
```

文件的权限会随内容复制。

```
$ python shutil_copy.py
```

```
BEFORE: []
```

```
AFTER: ['shutil_copy.py']
```

copy2() 的工作类似于 copy(), 不过复制到新文件的元数据中会包含访问和修改时间。

```
from shutil import *
import os
import time
def show_file_info(filename):
    stat_info = os.stat(filename)
    print '\tMode      :', stat_info.st_mode
    print '\tCreated   :', time.ctime(stat_info.st_ctime)
    print '\tAccessed  :', time.ctime(stat_info.st_atime)
    print '\tModified :', time.ctime(stat_info.st_mtime)

os.mkdir('example')
print 'SOURCE:'
show_file_info('shutil_copy2.py')
copy2('shutil_copy2.py', 'example')
print 'DEST:'
show_file_info('example/shutil_copy2.py')
```

这个新文件的所有特性都与原文件完全相同。

```
$ python shutil_copy2.py
```

```
SOURCE:
```

```
Mode      : 33188
Created   : Sat Dec  4 10:41:32 2010
Accessed  : Sat Dec  4 17:41:01 2010
Modified  : Sun Nov 14 09:40:36 2010
```

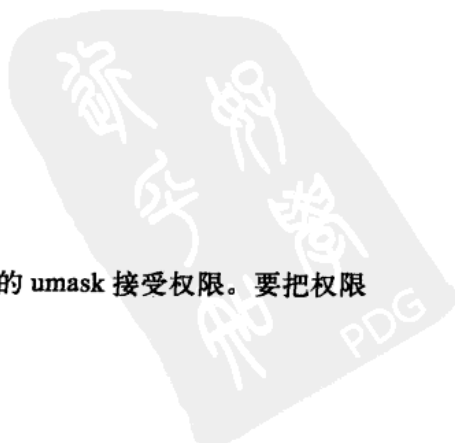
```
DEST:
```

```
Mode      : 33188
Created   : Sat Dec  4 17:41:01 2010
Accessed  : Sat Dec  4 17:41:01 2010
Modified  : Sun Nov 14 09:40:36 2010
```

6.5.2 复制文件元数据

默认地, 在 UNIX 下创建一个新文件时, 它会根据当前用户的 umask 接受权限。要把权限从一个文件复制到另一个文件, 可以使用 copymode()。

```
from shutil import *
from commands import *
import os
```



```

with open('file_to_change.txt', 'wt') as f:
    f.write('content')
os.chmod('file_to_change.txt', 0444)
print 'BEFORE:'
print getstatus('file_to_change.txt')
copymode('shutil_copymode.py', 'file_to_change.txt')
print 'AFTER : '
print getstatus('file_to_change.txt')

```

首先，创建一个要修改的文件。

```

#!/bin/sh
# Set up file needed by shutil_copymode.py
touch file_to_change.txt
chmod ugo+w file_to_change.txt

```

然后，运行示例脚本来改变权限。

```
$ python shutil_copymode.py
```

```

BEFORE:
-r--r--r--  1 dhellmann  dhellmann  7 Dec  4 17:41 file_to_change.txt
AFTER :
-rw-r--r--  1 dhellmann  dhellmann  7 Dec  4 17:41 file_to_change.txt

```

要复制文件的其他元数据，可以使用 `copystat()`。

```

from shutil import *
import os
import time

def show_file_info(filename):
    stat_info = os.stat(filename)
    print '\tMode      :', stat_info.st_mode
    print '\tCreated  :', time.ctime(stat_info.st_ctime)
    print '\tAccessed:', time.ctime(stat_info.st_atime)
    print '\tModified:', time.ctime(stat_info.st_mtime)

with open('file_to_change.txt', 'wt') as f:
    f.write('content')
os.chmod('file_to_change.txt', 0444)

print 'BEFORE:'
show_file_info('file_to_change.txt')
copystat('shutil_copystat.py', 'file_to_change.txt')
print 'AFTER:'
show_file_info('file_to_change.txt')

```

使用 `copystat()` 只会复制与文件关联的权限和日期。

```
$ python shutil_copystat.py
```

```

BEFORE:
Mode      : 33060
Created   : Sat Dec  4 17:41:01 2010
Accessed  : Sat Dec  4 17:41:01 2010
Modified  : Sat Dec  4 17:41:01 2010

AFTER:
Mode      : 33188
Created   : Sat Dec  4 17:41:01 2010
Accessed  : Sat Dec  4 17:41:01 2010
Modified  : Sun Nov 14 09:45:12 2010

```

6.5.3 处理目录树

shutil 包含 3 个函数用来处理目录树。要把一个目录从一个位置复制到另一个位置，可以使用 `copytree()`。这会递归遍历源目录树，将文件复制到目标。目标目录不能已存在。

注意：`copytree()` 的文档指出，应当把它看作是一个示例实现，而不是一个工具。可以考虑将当前这个实现作为起点，在真正使用之前，要让它更健壮，或者可以增加一些特性（如进度条）。

```

from shutil import *
from commands import *

print 'BEFORE:'
print getoutput('ls -rlast /tmp/example')
copytree('../shutil', '/tmp/example')
print '\nAFTER:'
print getoutput('ls -rlast /tmp/example')

```

`symlinks` 参数控制着符号链接作为链接复制还是作为文件复制。默认将内容复制到新文件。如果这个选项为 `true`，就会在目标树中创建新的符号链接。

```

$ python shutil_copytree.py

BEFORE:
ls: /tmp/example: No such file or directory

AFTER:
total 136
 8 -rwxr-xr-x  1 dhellmann  wheel   109 Oct 28 07:33 shutil_copymode.sh
 8 -rw-r--r--  1 dhellmann  wheel  1313 Nov 14 09:39 shutil_rmtree.py
 8 -rw-r--r--  1 dhellmann  wheel  1300 Nov 14 09:39 shutil_copyfile.py
 8 -rw-r--r--  1 dhellmann  wheel  1276 Nov 14 09:39 shutil_copy.py
 8 -rw-r--r--  1 dhellmann  wheel  1140 Nov 14 09:39 __init__.py
 8 -rw-r--r--  1 dhellmann  wheel  1595 Nov 14 09:40 shutil_copy2.py
 8 -rw-r--r--  1 dhellmann  wheel  1729 Nov 14 09:45 shutil_copystat.py
 8 -rw-r--r--  1 dhellmann  wheel    7 Nov 14 09:45 file_to_change.txt
 8 -rw-r--r--  1 dhellmann  wheel  1324 Nov 14 09:45 shutil_move.py
 8 -rw-r--r--  1 dhellmann  wheel   419 Nov 27 12:49 shutil_copymode.py
 8 -rw-r--r--  1 dhellmann  wheel  1331 Dec  1 21:51 shutil_copytree.py
 8 -rw-r--r--  1 dhellmann  wheel   816 Dec  4 17:39 shutil_copyfileobj.py
 8 -rw-r--r--  1 dhellmann  wheel    8 Dec  4 17:39 example.out

```

```

24 -rw-r--r-- 1 dhellmann wheel 9767 Dec 4 17:40 index.rst
8 -rw-r--r-- 1 dhellmann wheel 1300 Dec 4 17:41 shutil_copyfile.py.copy
0 drwxr-xr-x 3 dhellmann wheel 102 Dec 4 17:41 example
0 drwxrwxrwt 18 root wheel 612 Dec 4 17:41 ..
0 drwxr-xr-x 18 dhellmann wheel 612 Dec 4 17:41 .

```

要删除一个目录及其中的内容，可以使用 `rmtree()`。

```

from shutil import *
from commands import *

print 'BEFORE:'
print getoutput('ls -rlast /tmp/example')
rmtree('/tmp/example')
print 'AFTER:'
print getoutput('ls -rlast /tmp/example')

```

默认地，错误会作为异常产生，不过如果第二个参数为 `true`，就可以忽略这些异常。可以在第三个参数中提供一个特殊的错误处理函数。

```
$ python shutil_rmtree.py
```

```

BEFORE:
total 136
8 -rwxr-xr-x 1 dhellmann wheel 109 Oct 28 07:33 shutil_copymode.sh
8 -rw-r--r-- 1 dhellmann wheel 1313 Nov 14 09:39 shutil_rmtree.py
8 -rw-r--r-- 1 dhellmann wheel 1300 Nov 14 09:39 shutil_copyfile.py
8 -rw-r--r-- 1 dhellmann wheel 1276 Nov 14 09:39 shutil_copy.py
8 -rw-r--r-- 1 dhellmann wheel 1140 Nov 14 09:39 __init__.py
8 -rw-r--r-- 1 dhellmann wheel 1595 Nov 14 09:40 shutil_copy2.py
8 -rw-r--r-- 1 dhellmann wheel 1729 Nov 14 09:45 shutil_copystat.py
8 -rw-r--r-- 1 dhellmann wheel 7 Nov 14 09:45 file_to_change.txt
8 -rw-r--r-- 1 dhellmann wheel 1324 Nov 14 09:45 shutil_move.py
8 -rw-r--r-- 1 dhellmann wheel 419 Nov 27 12:49 shutil_copymode.py
8 -rw-r--r-- 1 dhellmann wheel 1331 Dec 1 21:51 shutil_copytree.py
8 -rw-r--r-- 1 dhellmann wheel 816 Dec 4 17:39 shutil_copyfileobj.py
8 -rw-r--r-- 1 dhellmann wheel 8 Dec 4 17:39 example.out
24 -rw-r--r-- 1 dhellmann wheel 9767 Dec 4 17:40 index.rst
8 -rw-r--r-- 1 dhellmann wheel 1300 Dec 4 17:41 shutil_copyfile.py.copy
0 drwxr-xr-x 3 dhellmann wheel 102 Dec 4 17:41 example
0 drwxrwxrwt 18 root wheel 612 Dec 4 17:41 ..
0 drwxr-xr-x 18 dhellmann wheel 612 Dec 4 17:41 .

AFTER:
ls: /tmp/example: No such file or directory

```

要把一个文件或目录从一个位置移动到另一个位置，可以使用 `move()`。

```

from shutil import *
from glob import glob

with open('example.txt', 'wt') as f:
    f.write('contents')

print 'BEFORE: ', glob('example*')
move('example.txt', 'example.out')
print 'AFTER : ', glob('example*')

```

其语义与 UNIX 命令 `mv` 类似。如果源和目标都在同一个文件系统中，则会重命名源文件。否则，源文件会复制到目标文件，然后将源文件删除。

```
$ python shutil_move.py
```

```
BEFORE: ['example.txt']
```

```
AFTER : ['example.out']
```

参见：

`shutil` (<http://docs.python.org/lib/module-shutil.html>) 这个模块的标准库文档。

6.6 mmap——内存映射文件

作用：建立内存映射文件而不是直接读取内容。

Python 版本：2.1 及以后版本

建立一个文件的内存映射将使用操作系统虚拟内存系统直接访问文件系统中的数据，而不是使用常规的 I/O 函数。内存映射通常可以提高 I/O 性能，因为使用内存映射时，不会对每个访问都有一个单独的系统调用，而且不需要在缓冲区之间复制数据——内核和用户应用都会直接访问内存。

内存映射文件可以看作是可修改的字符串或类文件对象，这取决于具体的需要。映射文件支持一般的文件 API 方法，如 `close()`、`flush()`、`read()`、`readline()`、`seek()`、`tell()` 和 `write()`。它还支持字符串 API，提供分片等特性以及类似 `find()` 的方法。

下面的所有示例都会使用文本文件 `lorem.txt`，其中包含一些 Lorem Ipsum。为便于参考，这里给出这个文件的文本。

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec
egestas, enim et consectetur ullamcorper, lectus ligula rutrum leo,
a elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque vel
arcu. Vivamus purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra fringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
mauris in nibh placerat egestas. Suspendisse potenti. Mauris
massa. Ut eget velit auctor tortor blandit sollicitudin. Suspendisse
imperdiet justo.
```

注意：UNIX 和 Windows 中 `mmap()` 的参数和行为有所差别。这里不会全面讨论这些差别。有关的更多细节，可以参考标准库文档。

6.6.1 读文件

使用 `mmap()` 函数可以创建一个内存映射文件。第一个参数是文件描述符，可以来自 `file` 对象的 `fileno()` 方法，或者来自 `os.open()`。调用者在调用 `mmap()` 之前负责打开文件，不再需要文件时要负责将其关闭。

`mmap()` 的第二个参数是要映射的文件部分的大小（以字节为单位）。如果这个值为 0，则映射整个文件。如果大小大于文件的当前大小，则会扩展该文件。

注意：Windows 不支持创建长度为 0 的映射。

这两个平台都支持一个可选的关键字参数 `access`。使用 `ACCESS_READ` 表示只读访问，`ACCESS_WRITE` 表示“写通过”（write-through），即对内存的赋值直接写入文件，或者 `ACCESS_COPY` 表示“写时复制”（copy-on-write），对内存的赋值不会写至文件。

```
import mmap
import contextlib

with open('lorem.txt', 'r') as f:
    with contextlib.closing(mmap.mmap(f.fileno(), 0,
                                      access=mmap.ACCESS_READ)
                          ) as m:
        print 'First 10 bytes via read :', m.read(10)
        print 'First 10 bytes via slice:', m[:10]
        print '2nd 10 bytes via read :', m.read(10)
```

文件指针会跟踪通过一个分片操作访问的最后一个字节。在这个例子中，第一次读之后，指针向前移动 10 个字节。然后由分片操作将指针重置回到文件的起始位置，并由于分片使指针再次向前移动 10 个字节。分片操作之后，再调用 `read()` 会给出文件的第 11~20 字节。

```
$ python mmap_read.py
```

```
First 10 bytes via read : Lorem ipsu
First 10 bytes via slice: Lorem ipsu
2nd 10 bytes via read : m dolor si
```

6.6.2 写文件

要建立内存映射文件来接收更新，映射之前首先要使用模式 `'r+'`（而不是 `'w'`）打开文件以便完成追加。然后可以使用任何改变数据的 API 方法（例如 `write()`，或赋值到一个分片，等等）。

下面的例子使用了默认访问模式 `ACCESS_WRITE`，并赋值到一个分片，原地修改某一行的一部分。

```
import mmap
import shutil
import contextlib
```

```

# Copy the example file
shutil.copyfile('lorem.txt', 'lorem_copy.txt')

word = 'consectetuer'
reversed = word[::-1]
print 'Looking for      :', word
print 'Replacing with :', reversed

with open('lorem_copy.txt', 'r+') as f:
    with contextlib.closing(mmap.mmap(f.fileno(), 0)) as m:
        print 'Before:'
        print m.readline().rstrip()
        m.seek(0) # rewind

        loc = m.find(word)
        m[loc:loc+len(word)] = reversed
        m.flush()

        m.seek(0) # rewind
        print 'After :'
        print m.readline().rstrip()

        f.seek(0) # rewind
        print 'File  :'
        print f.readline().rstrip()

```

内存和文件中第一行中间的单词“consectetuer”将被替换。

```
$ python mmap_write_slice.py
```

```

Looking for      : consectetuer
Replacing with : reutetcesnoc
Before:
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
After :
Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit. Donec
File  :
Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit. Donec

```

复制模式

使用访问设置 ACCESS_COPY 时不会将修改写入磁盘的文件。

```

import mmap
import shutil
import contextlib

# Copy the example file
shutil.copyfile('lorem.txt', 'lorem_copy.txt')

```



```

word = 'consectetuer'
reversed = word[::-1]

with open('lorem_copy.txt', 'r+') as f:
    with contextlib.closing(mmap.mmap(f.fileno(), 0,
                                      access=mmap.ACCESS_COPY)
                          ) as m:
        print 'Memory Before:'
        print m.readline().rstrip()
        print 'File Before  :'
        print f.readline().rstrip()
        print

        m.seek(0) # rewind
        loc = m.find(word)
        m[loc:loc+len(word)] = reversed

        m.seek(0) # rewind
        print 'Memory After :'
        print m.readline().rstrip()

        f.seek(0)
        print 'File After  :'
        print f.readline().rstrip()

```

在这个例子中，必须单独地回转文件句柄和 mmap 句柄，因为这两个对象的内部状态会单独维护。

```
$ python mmap_write_copy.py
```

```

Memory Before:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec
File Before  :
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec

Memory After :
Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit. Donec
File After   :
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec

```

6.6.3 正则表达式

由于内存映射文件就类似于一个字符串，因此也适用其他处理字符串的模块，如正则表达式。下面的例子会找出所有包含“nulla”的句子。

```

import mmap
import re
import contextlib

```

```

pattern = re.compile(r'(\.W+)?([^.]?nulla[^.]*?\.)',
                    re.DOTALL | re.IGNORECASE | re.MULTILINE)

with open('lorem.txt', 'r') as f:
    with contextlib.closing(mmap.mmap(f.fileno(), 0,
                                     access=mmap.ACCESS_READ)
                           ) as m:
        for match in pattern.findall(m):
            print match[1].replace('\n', ' ')

```

由于这个模式包含两个组，`findall()` 的返回值是一个元组序列。`print` 语句会找出匹配的句子，并用空格代替换行符，使各结果都打印在同一行上。

```
$ python mmap_regex.py
```

```

Nulla facilisi.
Nulla feugiat augue eleifend nulla.

```

参见：

`mmap` (<http://docs.python.org/lib/module-mmap.html>) 这个模块的标准库文档。

`os` (17.3 节) `os` 模块。

`contextlib` (3.4 节) 使用 `closing()` 函数为内存映射文件创建一个上下文管理器。

`re` (1.3 节) 正则表达式。

6.7 codecs——字符串编码和解码

作用：编码器和解码器，用于转换文本的不同表示。

Python 版本：2.1 及以后版本

`codecs` 模块提供了流接口和文件接口来转换数据。通常用于处理 Unicode 文本，不过也提供了其他编码来满足其他用途。

6.7.1 Unicode 入门

CPython 2.x 支持两种类型的字符串来处理文本数据。老式的 `str` 实例使用单个 8 位字节表示字符串字符（使用其 ASCII 码）。与之不同，`unicode` 串在内部作为一个 Unicode 码点（code point）序列来管理。码点值分别保存为两个或四个字节的序列，这取决于编译 Python 时指定的选项。`unicode` 和 `str` 都派生自一个公共基类，并支持类似的 API。

输出 `unicode` 串时，会使用某种标准机制编码，使得以后可以将这个字节序列重构为同样的文本串。编码值的字节不一定与码点值完全相同，编码只是定义了在一个值集之间转换的一种方式。读取 Unicode 数据时还需要知道编码，这样才能把接收到的字节转换为 `unicode` 类使用的内部表示。

西方语言最常用的编码是 UTF-8 和 UTF-16，这两种编码分别使用单字节和两字节值序列表示各个码点。对于其他语言，由于大多数字符都由超过两字节的码点表示，所以使用其他编码来存储可能更为高效。

参见：

关于 Unicode 的更多信息，请参考本节最后所列的参考资料。《The Python Unicode HOWTO》尤其有帮助。

编码

要了解编码，最好的方法就是采用不同方式对相同的串进行编码，并查看所生成的字节序列的区别。下面的例子使用以下函数格式化字节串，使之更易读。

```
import binascii

def to_hex(t, nbytes):
    """Format text t as a sequence of nbyte long values
    separated by spaces.
    """
    chars_per_item = nbytes * 2
    hex_version = binascii.hexlify(t)
    return ' '.join(
        hex_version[start:start + chars_per_item]
        for start in xrange(0, len(hex_version), chars_per_item)
    )

if __name__ == '__main__':
    print to_hex('abcdef', 1)
    print to_hex('abcdef', 2)
```

这个函数使用 binascii 得到输入字节串的十六进制表示，在返回这个值之前每隔 nbytes 字节就插入一个空格。

```
$ python codecs_to_hex.py
```

```
61 62 63 64 65 66
6162 6364 6566
```

第一个编码示例首先使用 unicode 类的原始表示打印文本 pi: π 。 π 字符替换为其 Unicode 码点的表达式 `\u03c0`。接下来的两行将这个字符串分别编码为 UTF-8 和 UTF-16，并显示编码得到的十六进制值。

```
from codecs_to_hex import to_hex

text = u'pi:  $\pi$ '

print 'Raw      :', repr(text)
print 'UTF-8    :', to_hex(text.encode('utf-8'), 1)
print 'UTF-16   :', to_hex(text.encode('utf-16'), 2)
```

对一个 unicode 串编码的结果是一个 str 对象。

```
$ python codecs_encodings.py
```

```
Raw      : u'pi: \u03c0'
UTF-8    : 70 69 3a 20 cf 80
UTF-16   : fffe 7000 6900 3a00 2000 c003
```

给定一个编码字节序列（作为一个 str 实例），decode() 方法会将其转换为码点，并作为一个 unicode 实例返回转换后的序列。

```
from codecs_to_hex import to_hex

text = u'pi:  $\pi$ '
encoded = text.encode('utf-8')
decoded = encoded.decode('utf-8')

print 'Original :', repr(text)
print 'Encoded   :', to_hex(encoded, 1), type(encoded)
print 'Decoded   :', repr(decoded), type(decoded)
```

选择使用哪一种编码不会改变输出类型。

```
$ python codecs_decode.py
```

```
Original : u'pi: \u03c0'
Encoded   : 70 69 3a 20 cf 80 <type 'str'>
Decoded   : u'pi: \u03c0' <type 'unicode'>
```

注意：解释器启动过程中（即加载 site 时）会设置默认编码。关于默认编码设置的描述，可以参考 sys 中的相关讨论。

6.7.2 处理文件

处理 I/O 操作时，编码和解码字符串尤其重要。不论是写至一个文件、一个套接字还是另一个流，数据都必须使用适当的编码。一般来讲，所有文本数据在读取时都需要从其字节表示解码，写时则需要从内部值编码为一种特定的表示。一个程序可以显式地编码和解码数据，但是取决于所用的编码，要确定是否已读取足够的字节以便充分解码数据，这一点可能并不容易。codecs 提供了一些类来管理数据的编码和解码，所以应用不需要做这个工作。

codecs 提供的最简单的接口可以用来替代内置 open() 函数。这个新版本的函数与内置函数的做法很相似，不过增加了两个参数来指定编码和所需的错误处理技术。

```
from codecs_to_hex import to_hex

import codecs
import sys
```

```

encoding = sys.argv[1]
filename = encoding + '.txt'

print 'Writing to', filename
with codecs.open(filename, mode='wt', encoding=encoding) as f:
    f.write(u'pi: \u03c0')

# Determine the byte grouping to use for to_hex()
nbytes = { 'utf-8':1,
            'utf-16':2,
            'utf-32':4,
            }.get(encoding, 1)

# Show the raw bytes in the file
print 'File contents:'
with open(filename, mode='rt') as f:
    print to_hex(f.read(), nbytes)

```

这个例子首先从一个 unicode 串开始（包含 π 的码点），并使用命令行上指定的编码将文本保存到一个文件。

```

$ python codecs_open_write.py utf-8

Writing to utf-8.txt
File contents:
70 69 3a 20 cf 80

$ python codecs_open_write.py utf-16

Writing to utf-16.txt
File contents:
fffe 7000 6900 3a00 2000 c003

$ python codecs_open_write.py utf-32

Writing to utf-32.txt
File contents:
fffe0000 70000000 69000000 3a000000 20000000 c0030000

```

用 `open()` 读数据很简单，但有一点要注意：必须提前知道编码，才能正确地建立解码器。尽管有些数据格式（如 XML）会指定编码作为文件的一部分，但是通常都要由应用来管理。`codecs` 只是取一个编码参数，并假设这个编码是正确的。

```

import codecs
import sys

encoding = sys.argv[1]
filename = encoding + '.txt'

```

```
print 'Reading from', filename
with codecs.open(filename, mode='rt', encoding=encoding) as f:
    print repr(f.read())
```

这个例子读取上一个程序创建的文件，并把得到的 unicode 对象的表示打印到控制台。

```
$ python codecs_open_read.py utf-8

Reading from utf-8.txt
u'pi: \u03c0'
$ python codecs_open_read.py utf-16

Reading from utf-16.txt
u'pi: \u03c0'

$ python codecs_open_read.py utf-32

Reading from utf-32.txt
u'pi: \u03c0'
```

6.7.3 字节序

在不同计算机系统之间传输数据时（可能直接复制一个文件，或者使用网络通信来完成传输），多字节编码（如 UTF-16 和 UTF-32）会引发一个问题。不同系统中使用的高字节和低字节的顺序不同。数据的这个特性，称为字节序（endianness），它依赖于硬件体系结构等因素，还取决于操作系统和应用开发人员做出的选择。通常没有办法提前知道给定的一组数据要使用哪一种字节序，所以多字节编码还包含一个字节序标志（byte-order marker, BOM），这个标志出现在编码输出的前几个字节。例如，UTF-16 定义 0xFFFE 和 0xFEFF 不是合法字符，而是用于指示字节序。codecs 定义了 UTF-16 和 UTF-32 所用字节序标志的相应常量。

```
import codecs
from codecs_to_hex import to_hex

for name in [ 'BOM', 'BOM_BE', 'BOM_LE',
              'BOM_UTF8',
              'BOM_UTF16', 'BOM_UTF16_BE', 'BOM_UTF16_LE',
              'BOM_UTF32', 'BOM_UTF32_BE', 'BOM_UTF32_LE',
              ]:
    print '{:12} : {}'.format(name, to_hex(getattr(codecs, name), 2))
```

取决于当前系统的固有字节序，BOM、BOM_UTF16 和 BOM_UTF32 会自动设置为适当的大端（big-endian）或小端（little-endian）值。

```
$ python codecs_bom.py

BOM          : fffe
```

```

BOM_BE      : feff
BOM_LE      : fffe
BOM_UTF8    : efbb bf
BOM_UTF16   : fffe
BOM_UTF16_BE : feff
BOM_UTF16_LE : fffe
BOM_UTF32   : fffe 0000
BOM_UTF32_BE : 0000 feff
BOM_UTF32_LE : fffe 0000

```

可以由 `codecs` 中的解码器自动检测和处理字节序，不过编码时也可以显式地指定字节序。

```

import codecs
from codecs_to_hex import to_hex

# Pick the nonnative version of UTF-16 encoding
if codecs.BOM_UTF16 == codecs.BOM_UTF16_BE:
    bom = codecs.BOM_UTF16_LE
    encoding = 'utf_16_le'
else:
    bom = codecs.BOM_UTF16_BE
    encoding = 'utf_16_be'

print 'Native order :', to_hex(codecs.BOM_UTF16, 2)
print 'Selected order:', to_hex(bom, 2)

# Encode the text.
encoded_text = u'pi: \u03c0'.encode(encoding)
print '{:14}: {}'.format(encoding, to_hex(encoded_text, 2))

with open('nonnative-encoded.txt', mode='wb') as f:
    # Write the selected byte-order marker. It is not included
    # in the encoded text because the byte order was given
    # explicitly when selecting the encoding.
    f.write(bom)
    # Write the byte string for the encoded text.
    f.write(encoded_text)

```

`codecs_bom_create_file.py` 首先得出内置的字节序，然后显式地使用候选形式，以便下一个例子可展示读取时自动检测字节序。

```
$ python codecs_bom_create_file.py
```

```

Native order : fffe
Selected order: feff
utf_16_be      : 0070 0069 003a 0020 03c0

```

`codecs_bom_detection.py` 打开文件时没有指定字节序，所以解码器会使用文件前两个字节中的 BOM 值来确定字节序。

```

import codecs
from codecs_to_hex import to_hex

# Look at the raw data
with open('nonnative-encoded.txt', mode='rb') as f:
    raw_bytes = f.read()

print 'Raw      :', to_hex(raw_bytes, 2)

# Reopen the file and let codecs detect the BOM
with codecs.open('nonnative-encoded.txt',
                 mode='rt',
                 encoding='utf-16',
                 ) as f:
    decoded_text = f.read()

print 'Decoded:', repr(decoded_text)

```

由于文件的前两个字节用于字节序检测，所以它们并不包含在 `read()` 返回的数据中。

```
$ python codecs_bom_detection.py
```

```

Raw      : feff 0070 0069 003a 0020 03c0
Decoded: u'pi: \u03c0'

```

6.7.4 错误处理

前面几节指出，读写 Unicode 文件时需要知道所使用的编码。正确地设置编码很重要，这有两个原因。如果读文件时未能正确地配置编码，就无法正确地解释数据，数据则有可能被破坏或者无法解码。并不是所有 Unicode 字符都可以采用所有编码表示，所以如果写文件时使用了错误的编码，就会生成一个错误，数据也可能丢失。

类似于 `unicode` 的 `encode()` 方法和 `str` 的 `decode()` 方法，`codecs` 也使用了同样的 5 个错误处理选项，如表 6.1 所列。

表 6.1 Codec 错误处理模式

错误模式	描述
<code>strict</code>	如果数据无法转换，产生一个异常
<code>replace</code>	将无法编码的数据替换为一个特殊的标志字符
<code>ignore</code>	跳过数据
<code>xmlcharrefreplace</code>	XML 字符（仅适用编码）
<code>backslashreplace</code>	转义序列（仅适用编码）

编码错误

最常见的错误条件是向一个 ASCII 输出流（如一个常规文件或 `sys.stdout`）写 Unicode 数据时接收到一个 `UnicodeEncodeError`。以下示例程序可以用来试验不同的错误处理模式。

```
import codecs
import sys

error_handling = sys.argv[1]

text = u'pi: \u03c0'

try:
    # Save the data, encoded as ASCII, using the error
    # handling mode specified on the command line.
    with codecs.open('encode_error.txt', 'w',
                    encoding='ascii',
                    errors=error_handling) as f:
        f.write(text)

except UnicodeEncodeError, err:
    print 'ERROR:', err
else:
    # If there was no error writing to the file,
    # show what it contains.
    with open('encode_error.txt', 'rb') as f:
        print 'File contents:', repr(f.read())
```

要确保一个应用显式地为所有 I/O 操作设置正确的编码，`strict` 模式是最安全的，但是产生一个异常时，这种模式可能导致程序崩溃。

```
$ python codecs_encode_error.py strict
```

```
ERROR: 'ascii' codec can't encode character u'\u03c0' in position 4:
ordinal not in range(128)
```

另外一些错误模式更为灵活。例如，`replace` 确保不会产生任何错误，其代价是可能会丢失一些无法转换为所需编码的数据。`pi(π)` 的 Unicode 字符仍然无法用 ASCII 编码，但是采用这种错误处理模式时，并不是产生一个异常，而是会在输出中将这个字符替换为 `?`。

```
$ python codecs_encode_error.py replace
```

```
File contents: 'pi: ?'
```

要完全跳过有问题的数据，可以使用 `ignore`。无法编码的数据都会被丢弃。

```
$ python codecs_encode_error.py ignore
```

```
File contents: 'pi: '
```

还有两种无损的错误处理选项，这两种模式会用一个不同于编码的标准中定义的候选表示替换字符。`xmlcharrefreplace` 使用一个 XML 字符引用来完成替换（W3C 文档《XML Entity Definitions for Characters》（字符的 XML 实体定义）中指定了字符引用列表）。

```
$ python codecs_encode_error.py xmlcharrefreplace
```

```
File contents: 'pi: &#960;'
```

另一种无损的错误处理机制是 `backslashreplace`，它生成的输出格式类似于打印 `unicode` 对象的 `repr()` 时返回的值。`Unicode` 字符会替换为 `\u`，后面会跟有码点的十六进制值。

```
$ python codecs_encode_error.py backslashreplace
```

```
File contents: 'pi: \\u03c0'
```

解码错误

数据解码时也有可能遇到错误，特别是在使用了错误的编码时。

```
import codecs
import sys

from codecs_to_hex import to_hex

error_handling = sys.argv[1]

text = u'pi: \u03c0'
print 'Original      :', repr(text)

# Save the data with one encoding
with codecs.open('decode_error.txt', 'w', encoding='utf-16') as f:
    f.write(text)

# Dump the bytes from the file
with open('decode_error.txt', 'rb') as f:
    print 'File contents:', to_hex(f.read(), 1)

# Try to read the data with the wrong encoding
with codecs.open('decode_error.txt', 'r',
                 encoding='utf-8',
                 errors=error_handling) as f:
    try:
        data = f.read()
    except UnicodeDecodeError, err:
        print 'ERROR:', err
    else:
        print 'Read      :', repr(data)
```

与编码一样，如果不能正确地解码字节流，`strict` 错误处理模式会产生一个异常。在这种情

况下，产生 `UnicodeDecodeError` 的原因是尝试将 UTF-16 BOM 的部分转换为一个使用 UTF-8 解码器的字符。

```
$ python codecs_decode_error.py strict
```

```
Original      : u'pi: \u03c0'
File contents: ff fe 70 00 69 00 3a 00 20 00 c0 03
ERROR: 'utf8' codec can't decode byte 0xff in position 0: invalid
start byte
```

切换到 `ignore` 会导致解码器跳过不合法的字节。不过，结果仍然不是原来期望的，因为其中包含嵌入的 `null` 字节。

```
$ python codecs_decode_error.py ignore
```

```
Original      : u'pi: \u03c0'
File contents: ff fe 70 00 69 00 3a 00 20 00 c0 03
Read          : u'p\x00i\x00:\x00 \x00\x03'
```

采用 `replace` 模式时，非法的字节会替换为 `\ufffd`，这是官方的 Unicode 替换字符，看起来像是一个有黑色背景的菱形，其中包含一个白色的问号。

```
$ python codecs_decode_error.py replace
```

```
Original      : u'pi: \u03c0'
File contents: ff fe 70 00 69 00 3a 00 20 00 c0 03
Read          : u'\ufffd\ufffdp\x00i\x00:\x00 \x00\ufffd\x03'
```

6.7.5 标准输入和输出流

之所以会产生 `UnicodeEncodeError` 异常，最常见的原因是代码中试图将 `unicode` 数据打印到控制台或一个 UNIX 管道，而 `sys.stdout` 未配置一个编码。

```
import codecs
import sys

text = u'pi: π'
# Printing to stdout may cause an encoding error
print 'Default encoding:', sys.stdout.encoding
print 'TTY:', sys.stdout.isatty()
print text
```

标准 I/O 通道默认编码存在的问题可能很难调试。这是因为，向控制台打印输出时，程序通常仍能如期工作，但是作为管道的一部分而且输出包括超出 ASCII 范围的 Unicode 字符时，则会导致一个编码错误。这种行为差异是 Python 的初始化代码造成的，它只在通道连接到一个终端时 (`isatty()` 返回 `True`) 才会为各个标准 I/O 通道设置默认编码。如果没有终端，Python 则假设程序会显式地配置编码，因此不再设置 I/O 通道的编码。

```
$ python codecs_stdout.py

Default encoding: utf-8
TTY: True
pi:  $\pi$ 

$ python codecs_stdout.py | cat -

Default encoding: None
TTY: False
Traceback (most recent call last):
  File "codecs_stdout.py", line 18, in <module>
    print text
UnicodeEncodeError: 'ascii' codec can't encode character
u'\u03c0' in position 4: ordinal not in range(128)
```

要显式地设置标准输出通道的编码，可以使用 `getwriter()` 得到一个特定编码的流编码器类。实例化这个类，传入 `sys.stdout` 作为其惟一参数。

```
import codecs
import sys

text = u'pi:  $\pi$ '

# Wrap sys.stdout with a writer that knows how to handle encoding
# Unicode data.
wrapped_stdout = codecs.getwriter('UTF-8')(sys.stdout)
wrapped_stdout.write(u'Via write: ' + text + '\n')

# Replace sys.stdout with a writer
sys.stdout = wrapped_stdout

print u'Via print:', text
```

写至包装的 `sys.stdout` 时，将编码字节发送到 `stdout` 之前会通过一个编码器传递 Unicode 文本。替换 `sys.stdout` 意味着应用中完成标准输出打印所使用的代码都能利用这个编码书写器。

```
$ python codecs_stdout_wrapped.py

Via write: pi:  $\pi$ 
Via print: pi:  $\pi$ 
```

下一个要解决的问题是如何知道应当使用哪一个编码。根据位置、语言以及用户或系统配置，可能会有不同的适用编码，所以在程序中硬编码一个固定值不是一个好的想法。对于用户来说，如果需要设置输入和输出编码从而向各个程序传递显式参数，这也很烦人。幸运的是，对此有一种全局方法，可以使用 `locale` 得到一个合理的默认编码。

```

import codecs
import locale
import sys

text = u'pi:  $\pi$ '

# Configure locale from the user's environment settings.
locale.setlocale(locale.LC_ALL, '')

# Wrap stdout with an encoding-aware writer.
lang, encoding = locale.getdefaultlocale()
print 'Locale encoding      : ', encoding
sys.stdout = codecs.getwriter(encoding)(sys.stdout)

print 'With wrapped stdout:', text

```

函数 `locale.getdefaultlocale()` 会根据系统和用户的配置设置返回语言和首选编码，并采用一种 `getwriter()` 可用的格式。

```
$ python codecs_stdout_locale.py
```

```

Locale encoding      : UTF8
With wrapped stdout: pi:  $\pi$ 

```

处理 `sys.stdin` 时也需要设置编码。使用 `getreader()` 可以得到一个能够解码输入字节的阅读器。

```

import codecs
import locale
import sys

# Configure locale from the user's environment settings.
locale.setlocale(locale.LC_ALL, '')

# Wrap stdin with an encoding-aware reader.
lang, encoding = locale.getdefaultlocale()
sys.stdin = codecs.getreader(encoding)(sys.stdin)

print 'From stdin:'
print repr(sys.stdin.read())

```

从包装的句柄读取数据时会返回 `unicode` 对象而不是 `str` 实例。

```
$ python codecs_stdout_locale.py | python codecs_stdin.py
```

```

From stdin:
u'Locale encoding      : UTF8\nWith wrapped stdout: pi: \u03c0\n'

```

6.7.6 编码转换

尽管大多数应用都在内部处理 unicode 数据，将数据解码或编码作为 I/O 操作的一部分，但有些情况下，可能需要改变文件的编码而不固守这种中间数据格式。EncodedFile() 取一个使用某种编码的打开文件句柄，用一个类来包装这个文件句柄，有 I/O 操作时则会把数据转换为另一种编码。

```
from codecs_to_hex import to_hex

import codecs
from cStringIO import StringIO

# Raw version of the original data.
data = u'pi: \u03c0'

# Manually encode it as UTF-8.
utf8 = data.encode('utf-8')
print 'Start as UTF-8    :', to_hex(utf8, 1)

# Set up an output buffer, then wrap it as an EncodedFile.
output = StringIO()
encoded_file = codecs.EncodedFile(output, data_encoding='utf-8',
                                   file_encoding='utf-16')

encoded_file.write(utf8)

# Fetch the buffer contents as a UTF-16 encoded byte string
utf16 = output.getvalue()
print 'Encoded to UTF-16:', to_hex(utf16, 2)

# Set up another buffer with the UTF-16 data for reading,
# and wrap it with another EncodedFile.
buffer = StringIO(utf16)
encoded_file = codecs.EncodedFile(buffer, data_encoding='utf-8',
                                   file_encoding='utf-16')

# Read the UTF-8 encoded version of the data.
recoded = encoded_file.read()
print 'Back to UTF-8    :', to_hex(recoded, 1)
```

这个例子显示了如何读写 EncodedFile() 返回的不同句柄。不论这个句柄用于读还是写，file_encoding 总是指示打开文件句柄所用的编码（作为第一个参数传入），data_encoding 值则指示通过 read() 和 write() 调用传递数据时所用的编码。

```
$ python codecs_encodedfile.py
```

```
Start as UTF-8    : 70 69 3a 20 cf 80
```

```
Encoded to UTF-16: fffe 7000 6900 3a00 2000 c003
Back to UTF-8    : 70 69 3a 20 cf 80
```

6.7.7 非 Unicode 编码

尽管之前大多数例子都使用 Unicode 编码，实际上 codecs 还可以用于很多其他数据转换。例如，Python 包含了 codecs 来处理 base-64、bzip2、ROT-13、ZIP 和其他数据格式。

```
import codecs
from cStringIO import StringIO

buffer = StringIO()
stream = codecs.getwriter('rot_13')(buffer)

text = 'abcdefghijklmnopqrstuvwxy'

stream.write(text)
stream.flush()

print 'Original:', text
print 'ROT-13  :', buffer.getvalue()
```

如果转换可以表述为有单个输入参数的函数，并返回一个字节或 Unicode 串，这样的转换都可以注册为一个 codec。

```
$ python codecs_rot13.py

Original: abcdefghijklmnopqrstuvwxy
ROT-13  : nopqrstuvwxyzabcdefghijklm
```

使用 codecs 包装一个数据流，可以提供比直接使用 zlib 更简单的接口。

```
import codecs
from cStringIO import StringIO

from codecs_to_hex import to_hex

buffer = StringIO()
stream = codecs.getwriter('zlib')(buffer)
text = 'abcdefghijklmnopqrstuvwxy\n' * 50

stream.write(text)
stream.flush()

print 'Original length :', len(text)
compressed_data = buffer.getvalue()
print 'ZIP compressed  :', len(compressed_data)

buffer = StringIO(compressed_data)
```



```

stream = codecs.getreader('zlib')(buffer)

first_line = stream.readline()
print 'Read first line :', repr(first_line)

uncompressed_data = first_line + stream.read()
print 'Uncompressed      :', len(uncompressed_data)
print 'Same              :', text == uncompressed_data

```

并不是所有压缩或编码系统都支持使用 `readline()` 或 `read()` 通过流接口读取数据的一部分，因为这需要找到压缩段的末尾来实现解压缩。如果一个程序无法在内存中保存整个解压缩的数据集，可以使用压缩库的增量访问特性，而不是 `codecs`。

```
$ python codecs_zlib.py
```

```

Original length : 1350
ZIP compressed  : 48
Read first line : 'abcdefghijklmnopqrstuvwxyz\n'
Uncompressed    : 1350
Same            : True

```

6.7.8 增量编码

目前提供的一些编码（特别是 `bz2` 和 `zlib`）在处理数据流时可能会显著改变数据流的长度。对于大的数据集，这些编码采用增量方式可以更好地处理，即一次只处理一个小数据块。`IncrementalEncoder` 和 `IncrementalDecoder` API 就设计用来完成这个任务。

```

import codecs
import sys
from codecs_to_hex import to_hex

text = 'abcdefghijklmnopqrstuvwxyz\n'
repetitions = 50

print 'Text length :', len(text)
print 'Repetitions :', repetitions
print 'Expected len:', len(text) * repetitions

# Encode the text several times to build up a large amount of data
encoder = codecs.getincrementalencoder('bz2')()
encoded = []

print
print 'Encoding:',
for i in range(repetitions):
    en_c = encoder.encode(text, final = (i==repetitions-1))
    if en_c:
        print '\nEncoded : {} bytes'.format(len(en_c))

```



```

        encoded.append(en_c)
    else:
        sys.stdout.write('.')

bytes = ''.join(encoded)
print
print 'Total encoded length:', len(bytes)
print

# Decode the byte string one byte at a time
decoder = codecs.getincrementaldecoder('bz2')()
decoded = []

print 'Decoding:',
for i, b in enumerate(bytes):
    final= (i+1) == len(text)
    c = decoder.decode(b, final)
    if c:
        print '\nDecoded : {} characters'.format(len(c))
        print 'Decoding:',
        decoded.append(c)
    else:
        sys.stdout.write('.')
print
restored = u''.join(decoded)

print
print 'Total uncompressed length:', len(restored)

```

每次将数据传递到编码器或解码器时，其内部状态会更新。状态一致时（按照 codec 的定义），会返回数据并重置状态。在此之前，`encode()` 或 `decode()` 调用并不返回任何数据。传入最后一部分数据时，参数 `final` 应当设置为 `True`，这样 codec 就能知道需要刷新输出所有余下的缓冲数据。

```
$ python codecs_incremental_bz2.py
```

```
Text length : 27
Repetitions : 50
Expected len: 1350
```

```
Encoding:.....
Encoded : 99 bytes
```

```
Total encoded length: 99
```

```
Decoding:.....
.....
```

```
Decoded : 1350 characters
Decoding:.....

Total uncompressed length: 1350
```

6.7.9 Unicode 数据和网络通信

类似于标准输入和输出文件描述符，网络套接字也是字节流，因此将 Unicode 数据写至一个套接字之前必须先编码为字节。以下服务器会把接收到的数据回送到发送者。

```
import sys
import SocketServer

class Echo(SocketServer.BaseRequestHandler):
    def handle(self):
        # Get some bytes and echo them back to the client.
        data = self.request.recv(1024)
        self.request.send(data)
        return

if __name__ == '__main__':
    import codecs
    import socket
    import threading

    address = ('localhost', 0) # let the kernel assign a port
    server = SocketServer.TCPServer(address, Echo)
    ip, port = server.server_address # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    # WRONG: Not encoded first!
    text = u'pi:  $\pi$ '
    len_sent = s.send(text)

    # Receive a response
    response = s.recv(len_sent)
    print repr(response)
```

```
# Clean up
s.close()
server.socket.close()
```

可以在每个 `send()` 调用之前对数据显式编码，不过如果少一个 `send()` 调用就会导致一个编码错误。

```
$ python codecs_socket_fail.py
Traceback (most recent call last):
File "codecs_socket_fail.py", line 43, in <module>
    len_sent = s.send(text)
UnicodeEncodeError: 'ascii' codec can't encode character
u'\u03c0' in position 4: ordinal not in range(128)
```

可以使用 `makefile()` 得到套接字的一个类文件句柄，然后用一个基于流的阅读器或书写器包装这个句柄，这意味着 Unicode 串传入和传出套接字时会被编码。

```
import sys
import SocketServer
```

```
class Echo(SocketServer.BaseRequestHandler):
```

```
    def handle(self):
        # Get some bytes and echo them back to the client. There is
        # no need to decode them, since they are not used.
        data = self.request.recv(1024)
        self.request.send(data)
        return
```

```
class PassThrough(object):
```

```
    def __init__(self, other):
        self.other = other

    def write(self, data):
        print 'Writing :', repr(data)
        return self.other.write(data)

    def read(self, size=-1):
        print 'Reading :',
        data = self.other.read(size)
        print repr(data)
        return data

    def flush(self):
```



```

        return self.other.flush()

    def close(self):
        return self.other.close()
if __name__ == '__main__':
    import codecs
    import socket
    import threading

    address = ('localhost', 0) # let the kernel assign a port
    server = SocketServer.TCPServer(address, Echo)
    ip, port = server.server_address # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Wrap the socket with a reader and writer.
    read_file = s.makefile('r')
    incoming = codecs.getreader('utf-8')(PassThrough(read_file))
    write_file = s.makefile('w')
    outgoing = codecs.getwriter('utf-8')(PassThrough(write_file))

    # Send the data
    text = u'pi:  $\pi$ '
    print 'Sending :', repr(text)
    outgoing.write(text)
    outgoing.flush()

    # Receive a response
    response = incoming.read()
    print 'Received:', repr(response)

    # Clean up
    s.close()
    server.socket.close()

```

这个例子使用 `PassThrough` 来展示数据在发送之前会进行编码，另外客户端接收响应之后会进行解码。

```
$ python codecs_socket.py
```

```
Sending : u'pi: \u03c0'
```

```

Writing : 'pi: \xcf\x80'
Reading : 'pi: \xcf\x80'
Received: u'pi: \u03c0'

```

6.7.10 定义定制编码

由于 Python 已经提供了大量标准编码，所以一般应用不太可能需要定义定制的编码器或解码器。不过，如果确实有必要，codecs 中包含有很多基类，使你能更容易地定义定制编码。

第一步是了解编码描述的转换性质。下面的例子将使用一个“invertcaps”编码，它把大写字母转换为小写，把小写字母转换为大写。下面是一个编码函数的简单定义，它会对一个输入字符串完成这个转换：

```

import string

def invertcaps(text):
    """Return new string with the case of all letters switched.
    """
    return ''.join( c.upper() if c in string.ascii_lowercase
                    else c.lower() if c in string.ascii_uppercase
                    else c
                    for c in text
                )

if __name__ == '__main__':
    print invertcaps('ABC.def')
    print invertcaps('abc.DEF')

```

在这里，编码器和解码器都是同一个函数（与 ROT-13 类似）。

```
$ python codecs_invertcaps.py
```

```

abc.DEF
ABC.def

```

尽管很容易理解，但这个实现效率不高，特别是对于非常大的文本串。幸运的是，codecs 包含一些辅助函数可以根据字符映射（character map）创建编码，如 invertcaps 编码。字符映射编码由两个字典构成。编码映射（encoding map）将输入串的字符值转换为输出中的字节值，解码映射（decoding map）则相反。首先创建解码映射，然后使用 make_encoding_map() 把它转换为一个编码映射。C 函数 charmap_encode() 和 charmap_decode() 可以使用这些映射高效地转换输入数据。

```

import codecs
import string

# Map every character to itself
decoding_map = codecs.make_identity_dict(range(256))

```

```

# Make a list of pairs of ordinal values for the lower and uppercase
# letters
pairs = zip([ ord(c) for c in string.ascii_lowercase],
            [ ord(c) for c in string.ascii_uppercase])

# Modify the mapping to convert upper to lower and lower to upper.
decoding_map.update( dict( (upper, lower)
                           for (lower, upper)
                           in pairs
                           )
                    )
decoding_map.update( dict( (lower, upper)
                           for (lower, upper)
                           in pairs
                           )
                    )

# Create a separate encoding map.
encoding_map = codecs.make_encoding_map(decoding_map)

if __name__ == '__main__':
    print codecs.charmap_encode('abc.DEF', 'strict', encoding_map)
    print codecs.charmap_decode('abc.DEF', 'strict', decoding_map)
    print encoding_map == decoding_map

```

尽管 invertcaps 的编码和解码映射是一样的，但并不总是如此。make_encoding_map() 会检测哪些情况下多个输入字符编码为相同的输出字节，并把编码值替换为 None，将编码标记为未定义。

```
$ python codecs_invertcaps_charmap.py
```

```

('ABC.def', 7)
(u'ABC.def', 7)
True

```

字符映射编码器和解码器支持前面介绍的所有标准错误处理方法，所以不需要为支持这部分 API 做任何额外的工作。

```

import codecs
from codecs_invertcaps_charmap import encoding_map

text = u'pi: π'

for error in [ 'ignore', 'replace', 'strict' ]:
    try:
        encoded = codecs.charmap_encode(text, error, encoding_map)
    except UnicodeEncodeError, err:

```

```

        encoded = str(err)
    print '{:7}: {}'.format(error, encoded)

```

由于 π 的 Unicode 码点不在编码映射中，所以采用 strict 错误处理模式时会产生一个异常。

```
$ python codecs_invertcaps_error.py
```

```

ignore : ('PI: ', 5)
replace: ('PI: ?', 5)
strict : 'charmap' codec can't encode character u'\u03c0' in position
         4: character maps to <undefined>

```

定义了编码和解码映射之后，还需要建立一些额外的类，另外要注册编码。register() 向注册表增加一个搜索函数，这样当用户希望使用这种编码时，codecs 能够找到。这个搜索函数必须有一个字符串参数，其中包含编码名，如果它知道这个编码则返回一个 CodecInfo 对象，否则返回 None。

```

import codecs
import encodings

def search1(encoding):
    print 'search1: Searching for:', encoding
    return None
def search2(encoding):
    print 'search2: Searching for:', encoding
    return None

codecs.register(search1)
codecs.register(search2)

utf8 = codecs.lookup('utf-8')
print 'UTF-8:', utf8

try:
    unknown = codecs.lookup('no-such-encoding')
except LookupError, err:
    print 'ERROR:', err

```

可以注册多个搜索函数，每个搜索函数将依次调用，直到一个搜索函数返回一个 CodecInfo，或者所有搜索函数都已经调用。codecs 注册的内部搜索函数知道如何加载标准编码，如 encodings 的 UTF-8，所以这些编码名不会传递到定制搜索函数。

```
$ python codecs_register.py
```

```

UTF-8: <codecs.CodecInfo object for encoding utf-8 at 0x100d0f530>
search1: Searching for: no-such-encoding
search2: Searching for: no-such-encoding
ERROR: unknown encoding: no-such-encoding

```

搜索函数返回的 `CodecInfo` 实例告诉 `codecs` 如何使用所支持的各种不同机制来完成编码和解码，包括：无状态编码、增量式编码和流编码。`codecs` 包括一些基类来帮助建立字符映射编码。下面这个例子集成了所有内容，它会注册一个搜索函数，并返回为 `invertcaps` 编码配置的一个 `CodecInfo` 实例。

```
import codecs

from codecs_invertcaps_charmap import encoding_map, decoding_map

# Stateless encoder/decoder

class InvertCapsCodec(codecs.Codec):
    def encode(self, input, errors='strict'):
        return codecs.charmap_encode(input, errors, encoding_map)
    def decode(self, input, errors='strict'):
        return codecs.charmap_decode(input, errors, decoding_map)

# Incremental forms

class InvertCapsIncrementalEncoder(codecs.IncrementalEncoder):
    def encode(self, input, final=False):
        data, nbytes = codecs.charmap_encode(input,
                                              self.errors,
                                              encoding_map)
        return data

class InvertCapsIncrementalDecoder(codecs.IncrementalDecoder):
    def decode(self, input, final=False):
        data, nbytes = codecs.charmap_decode(input,
                                              self.errors,
                                              decoding_map)
        return data

# Stream reader and writer

class InvertCapsStreamReader(InvertCapsCodec, codecs.StreamReader):
    pass

class InvertCapsStreamWriter(InvertCapsCodec, codecs.StreamWriter):
    pass

# Register the codec search function

def find_invertcaps(encoding):
    """Return the codec for 'invertcaps'."""
    if encoding == 'invertcaps':
```



```

    return codecs.CodecInfo(
        name='invertcaps',
        encode=InvertCapsCodec().encode,
        decode=InvertCapsCodec().decode,
        incrementalencoder=InvertCapsIncrementalEncoder,
        incrementaldecoder=InvertCapsIncrementalDecoder,
        streamreader=InvertCapsStreamReader,
        streamwriter=InvertCapsStreamWriter,
    )
return None
codecs.register(find_invertcaps)

if __name__ == '__main__':

    # Stateless encoder/decoder
    encoder = codecs.getencoder('invertcaps')
    text = 'abc.DEF'
    encoded_text, consumed = encoder(text)
    print 'Encoded "{}" to "{}", consuming {} characters'.format(
        text, encoded_text, consumed)

    # Stream writer
    import sys
    writer = codecs.getwriter('invertcaps')(sys.stdout)
    print 'StreamWriter for stdout: ',
    writer.write('abc.DEF')
    print

    # Incremental decoder
    decoder_factory = codecs.getincrementaldecoder('invertcaps')
    decoder = decoder_factory()
    decoded_text_parts = []
    for c in encoded_text:
        decoded_text_parts.append(decoder.decode(c, final=False))
    decoded_text_parts.append(decoder.decode('', final=True))
    decoded_text = ''.join(decoded_text_parts)
    print 'IncrementalDecoder converted "{}" to "{}"'.format(
        encoded_text, decoded_text)

```

无状态编码器 / 解码器的基类是 `Codec`，要用新实现覆盖 `encode()` 和 `decode()`（在这里分别调用了 `charmap_encode()` 和 `charmap_decode()`）。这些方法必须分别返回一个 `tuple`，其中包含转换的数据和利用过的输入字节或字符数。`charmap_encode()` 和 `charmap_decode()` 已经返回了这个信息，所以很方便。

`IncrementalEncoder` 和 `IncrementalDecoder` 可以作为增量式编码接口的基类。增量类的 `encode()` 和 `decode()` 方法定义为只返回具体的转换数据。有关缓冲的所有信息都作为内部状

态维护。invertcaps 编码不需要缓冲数据（它使用一种一对一映射）。如果编码根据所处理的数据会生成数量不同的输出，如压缩算法，对于这些编码，BufferedIncrementalEncoder 和 BufferedIncrementalDecoder 作为基类更为适合，因为它们可以管理输入中未处理的部分。

StreamReader 和 StreamWriter 也需要 encode() 和 decode() 方法，而且因为它们往往返回与 Codec 相应方法同样的值，所以实现时可以使用多重继承。

```
$ python codecs_invertcaps_register.py
```

```
Encoded "abc.DEF" to "ABC.def", consuming 7 characters
StreamWriter for stdout: ABC.def
IncrementalDecoder converted "ABC.def" to "abc.DEF"
```

参见：

codecs (<http://docs.python.org/library/codecs.html>) 这个模块的标准库文档。

locale (15.2 节) 访问和管理基于本地化的配置设置和行为。

io (<http://docs.python.org/library/io.html>) io 模块也包括处理编码和解码的文件和流包装器。

SocketServer (11.3 节) 要了解一个更详细的回应服务器例子，请参见 SocketServer 模块。

encodings 标准库中的一个包，其中包含 Python 提供的编码器/解码器实现。

PEP 100 (www.python.org/dev/peps/pep-0100) Python Unicode 集成 PEP。

Unicode HOWTO (<http://docs.python.org/howto/unicode>) Python 2.x 中使用 Unicode 的官方指南。

Python Unicode Objects (<http://effbot.org/zone/unicode-objects.htm>) Fredrik Lundh 撰写的一篇关于在 Python 2.0 中使用非 ASCII 字符的文章。

How to Use UTF-8 with Python (<http://evanjones.ca/python-utf8.html>) Evan Jones 编写的一个关于处理 Unicode 的快速指南，包括 XML 数据和字节序标志。

On the Goodness of Unicode (www.tbray.org/ongoing/When/200x/2003/04/06/Unicode) 对国际化和 Unicode 的介绍，作者是 Tim Bray。

On Character Strings (www.tbray.org/ongoing/When/200x/2003/04/13/Strings) 介绍编程语言中字符串处理历史的一篇文章，作者是 Tim Bray。

Characters vs. Bytes (www.tbray.org/ongoing/When/200x/2003/04/26/UTF) 这是 Tim Bray 的文章“essay on modern character string processing for computer programmers”的第一部分。这一部分涵盖了除 ASCII 字节以外其余格式文本的内存中表示。

Endianness (<http://en.wikipedia.org/wiki/Endianness>) 维基百科中对字节序的解释。

W3C XML Entity Definitions for Characters (www.w3.org/TR/xml-entity-names/) 如果字符引用无法采用一种编码表示，这里给出了这些字符引用 XML 表示的规范。

6.8 StringIO——提供类文件 API 的文本缓冲区

作用：使用一个类文件 API 处理文本缓冲区。

Python 版本：1.4 及以后版本

StringIO 提供了一种方便的做法，可以使用文件 API（read()、write() 等等）处理内存中的文本。有两种不同的实现。cStringIO 版本用 C 编写以提高速度，而 StringIO 用 Python 编写来提供可移植性。与其他一些字符串连接技术相比，使用 cStringIO 构造大字符串可以提供更好的性能。

示例

以下是使用 StringIO 缓冲区的一些标准例子：

```
# Find the best implementation available on this platform
try:
    from cStringIO import StringIO
except:
    from StringIO import StringIO

# Writing to a buffer
output = StringIO()
output.write('This goes into the buffer. ')
print '>>output, 'And so does this.'

# Retrieve the value written
print output.getvalue()

output.close() # discard buffer memory

# Initialize a read buffer
input = StringIO('Inital value for read buffer')
# Read from the buffer
print input.read()
```

这个例子使用了 read()，不过也可以用 readline() 和 readlines() 方法。StringIO 类还提供了一个 seek() 方法，读取文本时可以在缓冲区中跳转，如果使用一种前向解析算法，这对于回转很有用。

```
$ python stringio_examples.py
```

```
This goes into the buffer. And so does this.
```

```
Inital value for read buffer
```

参见：

StringIO (<http://docs.python.org/lib/module-StringIO.html>) 这个模块的标准库文档。

The StringIO module::: [www.effbot.org \(http://effbot.org/librarybook/stringio.htm\)](http://www.effbot.org/librarybook/stringio.htm) effbot 提供的 StringIO 例子。

Efficient String Concatenation in Python (www.skymind.com/%7Eocrow/python_string/) 分析了合并字符串的各种方法，并比较它们各自的优点。

6.9 fnmatch——UNIX 式 glob 模式匹配

作用：处理 UNIX 式文件名比较。

Python 版本：1.4 及以后版本

fnmatch 模块用于根据 glob 模式（如 UNIX shell 所用的模式）比较文件名。

6.9.1 简单匹配

fnmatch() 根据一个模式来比较一个文件名，并返回一个布尔值，指示二者是否匹配。如果操作系统使用一个区分大小写的文件系统，这个比较则是区分大小写的。

```
import fnmatch
import os
pattern = 'fnmatch_*.py'
print 'Pattern :', pattern
print

files = os.listdir('.')
for name in files:
    print 'Filename: %-25s %s' % \
        (name, fnmatch.fnmatch(name, pattern))
```

在这个例子中，这个模式会匹配所有以 fnmatch_ 开头并以 .py 结尾的文件。

```
$ python fnmatch_fnmatch.py
```

```
Pattern : fnmatch_*.py
```

```
Filename: __init__.py           False
Filename: fnmatch_filter.py     True
Filename: fnmatch_fnmatch.py    True
Filename: fnmatch_fnmatchcase.py True
Filename: fnmatch_translate.py  True
Filename: index.rst             False
```

要强制完成一个区分大小写的比较，而不论文件系统和操作系统如何设置，可以使用 fnmatchcase()。

```
import fnmatch
import os

pattern = 'FNMATCH_*.PY'
```

```

print 'Pattern :', pattern
print

files = os.listdir('.')

for name in files:
    print 'Filename: %-25s %s' % \
        (name, fnmatch.fnmatchcase(name, pattern))

```

由于用来测试这个程序的 OS X 系统使用的是区分大小写的文件系统，所以模式修改后无法匹配任何文件。

```

$ python fnmatch_fnmatchcase.py

Pattern : FNMATCH_*.PY

Filename: __init__.py           False
Filename: fnmatch_filter.py     False
Filename: fnmatch_fnmatch.py   False
Filename: fnmatch_fnmatchcase.py False
Filename: fnmatch_translate.py  False
Filename: index.rst             False

```

6.9.2 过滤

要测试一个文件名序列，可以使用 `filter()`，它会返回与模式参数匹配的文件名列表。

```

import fnmatch
import os
import pprint

pattern = 'fnmatch_*.py'
print 'Pattern :', pattern

files = os.listdir('.')

print
print 'Files   :'
pprint.pprint(files)

print
print 'Matches :'
pprint.pprint(fnmatch.filter(files, pattern))

```

在这个例子中，`filter()` 返回了与这一节有关的示例源文件的文件名列表。

```

$ python fnmatch_filter.py

Pattern : fnmatch_*.py

```



```
Files :
['__init__.py',
 'fnmatch_filter.py',
 'fnmatch_fnmatch.py',
 'fnmatch_fnmatchcase.py',
 'fnmatch_translate.py',
 'index.rst']

Matches :
['fnmatch_filter.py',
 'fnmatch_fnmatch.py',
 'fnmatch_fnmatchcase.py',
 'fnmatch_translate.py']
```

6.9.3 转换模式

在内部，fnmatch 将 glob 模式转换为一个正则表达式，并使用 re 模块来比较文件名和模式。translate() 函数就是将 glob 模式转换为正则表达式的公共 API。

```
import fnmatch

pattern = 'fnmatch_*.py'
print 'Pattern :', pattern
print 'Regex   :', fnmatch.translate(pattern)
```

要建立一个合法的表达式，需要对一些字符进行转义。

```
$ python fnmatch_translate.py
```

```
Pattern : fnmatch_*.py
Regex   : fnmatch\_\.\*\.\py\Z(?ms)
```

参见：

fnmatch (<http://docs.python.org/library/fnmatch.html>) 这个模块的标准库文档。

glob (6.2 节) glob 模块结合使用 fnmatch 匹配和 os.listdir()，来生成与模式匹配的文件和目录列表。

re (1.3 节) 正则表达式模式匹配。

6.10 dircache——缓存目录列表

作用：缓存目录列表，目录的修改时间改变时会更新。

Python 版本：1.4 及以后版本

dircache 模块从文件系统读取目录列表，并保存在内存中。

6.10.1 列出目录内容

dircache API 中的主要函数是 `listdir()`，这是 `os.listdir()` 的一个包装器。给定一个路径，每次调用 `dircache.listdir()` 时，这个函数会返回同样的 `list` 对象，除非目录的修改日期有改变。

```
import dircache

path = '.'
first = dircache.listdir(path)
second = dircache.listdir(path)

print 'Contents : '
for name in first:
    print ' ', name

print
print 'Identical:', first is second
print 'Equal    : ', first == second
```

有一点很重要，要认识到每次都会返回完全相同的 `list`，所以不能原地修改。

```
$ python dircache_listdir.py

Contents :
  __init__.py
  dircache_annotate.py
  dircache_listdir.py
  dircache_listdir_file_added.py
  dircache_reset.py
  index.rst
```

```
Identical: True
Equal    : True
```

如果目录的内容有改变，则会重新扫描。

```
import dircache
import os

path = '/tmp'
file_to_create = os.path.join(path, 'pymotw_tmp.txt')

# Look at the directory contents
first = dircache.listdir(path)

# Create the new file
open(file_to_create, 'wt').close()

# Rescan the directory
```



```

second = dircache.listdir(path)

# Remove the file we created
os.unlink(file_to_create)

print 'Identical :', first is second
print 'Equal      :', first == second
print 'Difference:', list(set(second) - set(first))

```

在这种情况下，由于出现了新文件，这就导致构造一个新的 list。

```
$ python dircache_listdir_file_added.py
```

```

Identical : False
Equal      : False
Difference: ['pymotw_tmp.txt']

```

还可以重置整个缓存，删除其内容，从而重新检查每一个路径。

```

import dircache

path = '/tmp'
first = dircache.listdir(path)
dircache.reset()
second = dircache.listdir(path)

print 'Identical :', first is second
print 'Equal      :', first == second
print 'Difference:', list(set(second) - set(first))

```

重置之后，会返回一个新的 list 实例。

```
$ python dircache_reset.py
```

```

Identical : False
Equal      : True
Difference: []

```

6.10.2 标注列表

dircache 模块还提供了另一个有趣的函数：annotate()，它会修改 list()（如 listdir() 返回的 list），向表示目录的名称末尾增加一个“/”。

```

import dircache
from pprint import pprint
import os

path = '../..'

contents = dircache.listdir(path)

```



```

annotated = contents[:]
dircache.annotate(path, annotated)

fmt = '%25s\t%25s'

print fmt % ('ORIGINAL', 'ANNOTATED')
print fmt % (('-' * 25,)*2)
for o, a in zip(contents, annotated):
    print fmt % (o, a)

```

遗憾的是，对于 Windows 用户，尽管 `annotate()` 使用 `os.path.join()` 来构造要测试的名称，但是它总是追加一个 “/”，而不是 `os.sep`。

```
$ python dircache_annotate.py
```

ORIGINAL	ANNOTATED
-----	-----
.DS_Store	.DS_Store
.hg	.hg/
.hgignore	.hgignore
.hgtags	.hgtags
LICENSE.txt	LICENSE.txt
MANIFEST.in	MANIFEST.in
PyMOTW	PyMOTW/
PyMOTW.egg-info	PyMOTW.egg-info/
README.txt	README.txt
bin	bin/
dist	dist/
module	module
motw	motw
output	output/
pavement.py	pavement.py
paver-minilib.zip	paver-minilib.zip
setup.py	setup.py
sitemap_gen_config.xml	sitemap_gen_config.xml
sphinx	sphinx/
structure	structure/
trace.txt	trace.txt
utils	utils/

参见：

`dircache` (<http://docs.python.org/library/dircache.html>) 这个模块的标准库文档。

6.11 filecmp——比较文件

作用：比较文件系统中的文件和目录。

Python 版本: 2.1 及以后版本

filecmp 模块包含一些函数和一个类来比较文件系统中的文件和目录。

6.11.1 示例数据

以下讨论的例子使用了 filecmp_mkexamples.py 创建的一组测试文件。

```
import os

def mkfile(filename, body=None):
    with open(filename, 'w') as f:
        f.write(body or filename)
    return

def make_example_dir(top):
    if not os.path.exists(top):
        os.mkdir(top)
    curdir = os.getcwd()
    os.chdir(top)

    os.mkdir('dir1')
    os.mkdir('dir2')

    mkfile('dir1/file_only_in_dir1')
    mkfile('dir2/file_only_in_dir2')

    os.mkdir('dir1/dir_only_in_dir1')
    os.mkdir('dir2/dir_only_in_dir2')

    os.mkdir('dir1/common_dir')
    os.mkdir('dir2/common_dir')

    mkfile('dir1/common_file', 'this file is the same')
    mkfile('dir2/common_file', 'this file is the same')

    mkfile('dir1/not_the_same')
    mkfile('dir2/not_the_same')

    mkfile('dir1/file_in_dir1', 'This is a file in dir1')
    os.mkdir('dir2/file_in_dir1')

    os.chdir(curdir)
    return

if __name__ == '__main__':
    os.chdir(os.path.dirname(__file__) or os.getcwd())
    make_example_dir('example')
    make_example_dir('example/dir1/common_dir')
    make_example_dir('example/dir2/common_dir')
```

运行 `filecmp_mkexamples.py` 时，会在 `example` 目录下生成一个文件树：

```
$ find example
```

```
example
example/dir1
example/dir1/common_dir
example/dir1/common_dir/dir1
example/dir1/common_dir/dir1/common_dir
example/dir1/common_dir/dir1/common_file
example/dir1/common_dir/dir1/dir_only_in_dir1
example/dir1/common_dir/dir1/file_in_dir1
example/dir1/common_dir/dir1/file_only_in_dir1
example/dir1/common_dir/dir1/not_the_same
example/dir1/common_dir/dir2
example/dir1/common_dir/dir2/common_dir
example/dir1/common_dir/dir2/common_file
example/dir1/common_dir/dir2/dir_only_in_dir2
example/dir1/common_dir/dir2/file_in_dir1
example/dir1/common_dir/dir2/file_only_in_dir2
example/dir1/common_dir/dir2/not_the_same
example/dir1/common_file
example/dir1/dir_only_in_dir1
example/dir1/file_in_dir1
example/dir1/file_only_in_dir1
example/dir1/not_the_same
example/dir2
example/dir2/common_dir
example/dir2/common_dir/dir1
example/dir2/common_dir/dir1/common_dir
example/dir2/common_dir/dir1/common_file
example/dir2/common_dir/dir1/dir_only_in_dir1
example/dir2/common_dir/dir1/file_in_dir1
example/dir2/common_dir/dir1/file_only_in_dir1
example/dir2/common_dir/dir1/not_the_same
example/dir2/common_dir/dir2
example/dir2/common_dir/dir2/common_dir
example/dir2/common_dir/dir2/common_file
example/dir2/common_dir/dir2/dir_only_in_dir2
example/dir2/common_dir/dir2/file_in_dir1
example/dir2/common_dir/dir2/file_only_in_dir2
example/dir2/common_dir/dir2/not_the_same
example/dir2/common_file
example/dir2/dir_only_in_dir2
example/dir2/file_in_dir1
example/dir2/file_only_in_dir2
example/dir2/not_the_same
```



“common_dir” 目录下也有同样的目录结构，以便完成有趣的递归比较。

6.11.2 比较文件

cmp() 用于比较文件系统中的两个文件。

```
import filecmp

print 'common_file:',
print filecmp.cmp('example/dir1/common_file',
                  'example/dir2/common_file'),
print filecmp.cmp('example/dir1/common_file',
                  'example/dir2/common_file',
                  shallow=False)

print 'not_the_same:',
print filecmp.cmp('example/dir1/not_the_same',
                  'example/dir2/not_the_same'),
print filecmp.cmp('example/dir1/not_the_same',
                  'example/dir2/not_the_same',
                  shallow=False)

print 'identical:',
print filecmp.cmp('example/dir1/file_only_in_dir1',
                  'example/dir1/file_only_in_dir1'),
print filecmp.cmp('example/dir1/file_only_in_dir1',
                  'example/dir1/file_only_in_dir1',
                  shallow=False)
```

shallow 参数告诉 cmp() 除了文件的元数据外，是否还要查看文件的内容。默认情况下，会使用由 os.stat() 得到的信息完成一个浅比较，而不查看内容。对于同时创建的相同大小的文件，如果不比较其内容，会报告为相同。

```
$ python filecmp_cmp.py
```

```
common_file: True True
not_the_same: True False
identical: True True
```

如果非递归地比较两个目录中的一组文件，可以使用 cmpfiles()。参数是目录名和两个位置上要检查的文件列表。传入的公共文件列表应当只包含文件名（目录会导致匹配不成功），而且这些文件在两个位置上都应当出现。下一个例子显示了构造公共列表的一种简单方法。与 cmp() 一样，这个比较也可以取一个 shallow 标志。

```
import filecmp
import os

# Determine the items that exist in both directories
d1_contents = set(os.listdir('example/dir1'))
```

```

d2_contents = set(os.listdir('example/dir2'))
common = list(d1_contents & d2_contents)
common_files = [ f
    for f in common
    if os.path.isfile(os.path.join('example/dir1', f))
]
print 'Common files:', common_files

# Compare the directories
match, mismatch, errors = filecmp.cmpfiles('example/dir1',
                                           'example/dir2',
                                           common_files)

print 'Match      :', match
print 'Mismatch  :', mismatch
print 'Errors    :', errors

```

cmpfiles() 返回 3 个文件名列表，分别包含匹配的文件、不匹配的文件和不能比较的文件（由于权限问题或出于其他原因）。

```
$ python filecmp_cmpfiles.py
```

```

Common files: ['not_the_same', 'file_in_dir1', 'common_file']
Match      : ['not_the_same', 'common_file']
Mismatch   : ['file_in_dir1']
Errors     : []

```

6.11.3 比较目录

前面介绍的函数适合完成相对简单的比较。对于大目录树的递归比较或者完成更完整的分析，dircmp 类会更有用。在以下最简单的使用用例中，report() 会打印比较两个目录的报告。

```

import filecmp

filecmp.dircmp('example/dir1', 'example/dir2').report()

```

输出是一个纯文本报告，显示的结果只包括给定目录的内容，而不会递归比较其子目录。在这里，文件“not_the_same”被认为是相同的，因为并没有比较内容。无法让 dircmp 像 cmp() 那样比较文件的内容。

```
$ python filecmp_dircmp_report.py
```

```

diff example/dir1 example/dir2
Only in example/dir1 : ['dir_only_in_dir1', 'file_only_in_dir1']
Only in example/dir2 : ['dir_only_in_dir2', 'file_only_in_dir2']
Identical files : ['common_file', 'not_the_same']
Common subdirectories : ['common_dir']
Common funny cases : ['file_in_dir1']

```

要想完成更为详细的递归比较，可以使用 report_full_closure()：

```
import filecmp
```

```
filecmp.dircmp('example/dir1', 'example/dir2').report_full_closure()
```

输出将包括所有同级子目录的比较。

```
$ python filecmp_dircmp_report_full_closure.py
```

```
diff example/dir1 example/dir2
Only in example/dir1 : ['dir_only_in_dir1', 'file_only_in_dir1']
Only in example/dir2 : ['dir_only_in_dir2', 'file_only_in_dir2']
Identical files : ['common_file', 'not_the_same']
Common subdirectories : ['common_dir']
Common funny cases : ['file_in_dir1']

diff example/dir1/common_dir example/dir2/common_dir
Common subdirectories : ['dir1', 'dir2']

diff example/dir1/common_dir/dir2 example/dir2/common_dir/dir2
Identical files : ['common_file', 'file_only_in_dir2', 'not_the_same']
]
Common subdirectories : ['common_dir', 'dir_only_in_dir2', 'file_in_dir1']

diff example/dir1/common_dir/dir2/common_dir example/dir2/common_dir/dir2/common_dir

diff example/dir1/common_dir/dir2/dir_only_in_dir2 example/dir2/common_dir/dir2/dir_only_in_dir2

diff example/dir1/common_dir/dir2/file_in_dir1 example/dir2/common_dir/dir2/file_in_dir1

diff example/dir1/common_dir/dir1 example/dir2/common_dir/dir1
Identical files : ['common_file', 'file_in_dir1', 'file_only_in_dir1', 'not_the_same']
Common subdirectories : ['common_dir', 'dir_only_in_dir1']

diff example/dir1/common_dir/dir1/common_dir example/dir2/common_dir/dir1/common_dir

diff example/dir1/common_dir/dir1/dir_only_in_dir1 example/dir2/common_dir/dir1/dir_only_in_dir1
```

6.11.4 程序中使用差异

除了生成打印报告，`dircmp` 还能计算文件列表，可以在程序中直接使用。以下各个属性只

在请求时才计算，所以对于未用的数据，创建 `dircmp` 实例并不会带来开销。

```
import filecmp
import pprint
dc = filecmp.dircmp('example/dir1', 'example/dir2')
print 'Left:'
pprint.pprint(dc.left_list)

print '\nRight:'
pprint.pprint(dc.right_list)
```

所比较目录中包含的文件和子目录分别列在 `left_list` 和 `right_list` 中。

```
$ python filecmp_dircmp_list.py
```

```
Left:
['common_dir',
 'common_file',
 'dir_only_in_dir1',
 'file_in_dir1',
 'file_only_in_dir1',
 'not_the_same']
```

```
Right:
['common_dir',
 'common_file',
 'dir_only_in_dir2',
 'file_in_dir1',
 'file_only_in_dir2',
 'not_the_same']
```

可以向构造函数传入一个要忽略的名字列表（该列表中指定的名字将被忽略），对输入进行过滤。默认情况下，RCS、CVS 和 tags 等名字会被忽略。

```
import filecmp
import pprint

dc = filecmp.dircmp('example/dir1', 'example/dir2',
                    ignore=['common_file'])

print 'Left:'
pprint.pprint(dc.left_list)

print '\nRight:'
pprint.pprint(dc.right_list)
```

在这里，将把 “common_file” 从要比较的文件列表中去除。



```
$ python filecmp_dircmp_list_filter.py
```

```
Left:
['common_dir',
 'dir_only_in_dir1',
 'file_in_dir1',
 'file_only_in_dir1',
 'not_the_same']
```

```
Right:
['common_dir',
 'dir_only_in_dir2',
 'file_in_dir1',
 'file_only_in_dir2',
 'not_the_same']
```

两个输入目录中共有的文件名会保存在 `common`，各目录独有的文件会列在 `left_only` 和 `right_only` 中。

```
import filecmp
import pprint

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print 'Common:'
pprint.pprint(dc.common)

print '\nLeft:'
pprint.pprint(dc.left_only)

print '\nRight:'
pprint.pprint(dc.right_only)
```

“left” 目录是 `dircmp()` 的第一个参数，“right” 目录是第二个参数。

```
$ python filecmp_dircmp_membership.py
```

```
Common:
['not_the_same', 'common_file', 'file_in_dir1', 'common_dir']

Left:
['dir_only_in_dir1', 'file_only_in_dir1']

Right:
['dir_only_in_dir2', 'file_only_in_dir2']
```

公共成员可以进一步分解为文件、目录和“有趣”（funny）元素（两个目录中类型不同的内容，或者 `os.stat()` 指出错误的地方）。

```
import filecmp
import pprint
```



```

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print 'Common:'
pprint.pprint(dc.common)

print '\nDirectories:'
pprint.pprint(dc.common_dirs)

print '\nFiles:'
pprint.pprint(dc.common_files)

print '\nFunny:'
pprint.pprint(dc.common_funny)

```

在示例数据中，名为“file_in_dir1”的元素在一个目录中是一个文件，而在另一个目录中是一个子目录，所以它会出现在“有趣”列表中。

```
$ python filecmp_dircmp_common.py
```

```

Common:
['not_the_same', 'common_file', 'file_in_dir1', 'common_dir']

Directories:
['common_dir']

Files:
['not_the_same', 'common_file']

Funny:
['file_in_dir1']

```

文件之间的差别也可以做类似的划分。

```

import filecmp

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print 'Same      :', dc.same_files
print 'Different :', dc.diff_files
print 'Funny     :', dc.funny_files

```

文件 not_the_same 只通过 os.stat() 比较，并不检查内容，所以它会包含在 same_files 列表中。

```

$ python filecmp_dircmp_diff.py

Same      : ['not_the_same', 'common_file']
Different : []
Funny     : []

```

最后一点，子目录也会保存，从而能容易地完成递归比较。

```
import filecmp

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print 'Subdirectories:'
print dc.subdirs
```

属性 `subdirs` 是一个字典，它将目录名映射到新的 `dircmp` 对象。

```
$ python filecmp_dircmp_subdirs.py
```

```
Subdirectories:
{'common_dir': <filecmp.dircmp instance at 0x85da0>}
```

参见：

`filecmp` (<http://docs.python.org/library/filecmp.html>) 这个模块的标准库文档。

`Directories` (17.3.7 节) 使用 `os` 列出一个目录的内容。

`difflib` (1.4 节) 计算两个序列之间的差异。



第 ⑦ 章

数据持久存储与交换

要持久存储数据以供长期使用，这包括两个方面：在对象的内存中表示和存储格式之间来回转换数据，以及处理转换后数据的存储区。标准库包含很多模块，可以在不同情况下处理这两个方面。

有两个模块可以将对象转换为一种可以传输或存储的格式[这个过程称为串行化(serializing)]。最常用的是使用 pickle 完成持久存储，因为它可以与其他一些具体存储串行化数据的标准库模块集成，如 shelve。不过对基于 Web 的应用，json 则更为常用，因为它能更好地与现有 Web 服务存储工具集成。

一旦将内存中对象转换为一种可以保存的格式，下一步就是确定如何存储这个数据。如果数据不需要以某种方式索引，依序先后写入串行化对象的简单平面文件就很适用。Python 包括一组模块可以在一个简单的数据库中存储键-值对，需要索引查找时会使用某种变种 DBM 格式。

要利用 DBM 格式，最直接的方式就是 shelve。可以打开 shelve 文件，并通过一个类字典的 API 来访问。保存到数据库的对象会自动 pickle 并保存，而无须调用者做任何额外的工作。

不过 shelve 有一个缺点，使用默认接口时，没有办法预测将使用哪一个 DBM 格式，因为它会根据创建数据库的系统上有哪些可用的库来进行选择。如果应用不需要在库配置不同的主机之间共享数据库文件，那么选择哪种格式并不重要；不过，如果必须保证可移植性，可以使用这个模块中的某个类来确保选择一个特定的格式。

对于 Web 应用，由于这些应用处理的就是 JSON 格式的数据，因此使用 json 和 anydbm 可以提供另一种持久存储机制。直接使用 anydbm 会比 shelve 稍微多做一些工作，因为 DBM 数据库键和值必须是字符串，而且在数据库中访问值时不会自动地重新创建对象。

大多数 Python 发布版本都提供了 sqlite3 进程中关系数据库，可以采用比键-值对更复杂的组织来存储数据。它将数据库存储在内存中或者存储在一个本地文件中，所有访问都来自同一个进程，所以不存在网络通信延迟。sqlite3 的紧凑性使它尤其适合嵌入到桌面应用或 Web 应用的开发版本中。

还有一些模块可以用来解析定义更为形式化的格式，这对于在 Python 程序和用其他语言编写的应用之间交换数据非常有用。xml.etree.ElementTree 可以解析 XML 文档，为不同应用提供多种操作模式。除了解析工具，ElementTree 还包括一个接口可以由内存中的对象创建良构的 XML 文档。csv 模块可以读写表格数据（采用由电子表格或数据库应用生成的格式），这对于批量加载数据或者将数据从一种格式转换为另一种格式非常有用。

7.1 pickle——对象串行化

作用：对象串行化。

Python 版本：pickle 为 1.4 及以后版本，cPickle 为 1.5 及以后版本

pickle 模块实现了一个算法可以将一个任意的 Python 对象转换为一系列字节。这个过程也称为串行化对象。表示对象的字节流可以传输或存储，然后重新构造来创建有相同性质的新对象。

cPickle 模块实现了同样的算法，不过用 C 实现而不是 Python。它比 Python 实现要快数倍，所以通常会使用这个模块而不是纯 Python 实现。

警告： pickle 的文档明确指出它不提供任何安全保证。实际上，对数据解除 pickle 可以执行任意的代码。使用 pickle 完成进程间通信或数据存储时要当心，另外不要相信未得到安全验证的数据。可以参见 hmac 一节，其中有一个例子展示了采用一种安全方式来验证 pickle 数据来源。

7.1.1 导入

由于 cPickle 比 pickle 更快，所以通常首先会尝试导入 cPickle，并给定一个别名“pickle”，如果导入失败，则退而使用 pickle 中的内置 Python 实现。这说明，如果有更快的实现，程序总是倾向于使用更快的实现，否则才使用可移植的实现。

```
try:
    import cPickle as pickle
except:
    import pickle
```

C 和 Python 版本的 API 完全相同，数据可以在使用 C 和 Python 版本库的程序之间交换。

7.1.2 编码和解码字符串数据

第一个例子使用 dumps() 将一个数据结构编码为一个字符串，然后把这个字符串打印到控制台。它使用了一个完全由内置类型构成的数据结构。任何类的实例都可以 pickle，如下例所示。

```
try:
    import cPickle as pickle
except:
    import pickle
import pprint

data = [ { 'a': 'A', 'b': 2, 'c': 3.0 } ]
print 'DATA:',
pprint.pprint(data)
```

```
data_string = pickle.dumps(data)
print 'PICKLE: %r' % data_string
```

默认情况下, pickle 只包含 ASCII 字符。还有一种更高效的二进制 pickle 格式, 不过这里的所有例子都使用 ASCII 输出, 因为这样在打印时更容易理解。

```
$ python pickle_string.py
```

```
DATA:[{'a': 'A', 'b': 2, 'c': 3.0}]
PICKLE: "(lp1\n(dp2\nS'a'\nS'A'\nsS'c'\nF3\nsS'b'\nI2\nsa."
```

数据串行化后, 可以写到一个文件、套接字或者管道等等。之后可以读取这个文件, 将数据解除 pickle, 用同样的值构造一个新的对象。

```
try:
    import cPickle as pickle
except:
    import pickle
import pprint

data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE: ',
pprint.pprint(data1)

data1_string = pickle.dumps(data1)

data2 = pickle.loads(data1_string)
print 'AFTER : ',
pprint.pprint(data2)

print 'SAME? :', (data1 is data2)
print 'EQUAL?:', (data1 == data2)
```

新构造的对象等于原来的对象, 但并不是同一个对象。

```
$ python pickle_unpickle.py
```

```
BEFORE: [{'a': 'A', 'b': 2, 'c': 3.0}]
AFTER : [{'a': 'A', 'b': 2, 'c': 3.0}]
SAME? : False
EQUAL?: True
```

7.1.3 处理流

除了 dumps() 和 loads(), pickle 还提供了一些便利函数处理类文件的流。可以向一个流写多个对象, 然后从流读取这些对象, 而无需事先知道要写多少个对象或者这些对象有多大。

```
try:
    import cPickle as pickle
```

```

except:
    import pickle
    import pprint
    from StringIO import StringIO

class SimpleObject(object):

    def __init__(self, name):
        self.name = name
        self.name_backwards = name[::-1]
        return

data = []
data.append(SimpleObject('pickle'))
data.append(SimpleObject('cPickle'))
data.append(SimpleObject('last'))

# Simulate a file with StringIO
out_s = StringIO()

# Write to the stream
for o in data:
    print 'WRITING : %s (%s)' % (o.name, o.name_backwards)
    pickle.dump(o, out_s)
    out_s.flush()

# Set up a read-able stream
in_s = StringIO(out_s.getvalue())

# Read the data
while True:
    try:
        o = pickle.load(in_s)
    except EOFError:
        break
    else:
        print 'READ      : %s (%s)' % (o.name, o.name_backwards)

```

这个例子使用两个 StringIO 缓冲区来模拟流。第一个缓冲区接收 pickle 的对象，将其值传入到第二个缓冲区，load() 将读取这个缓冲区。简单的数据库格式也可以使用 pickle 来存储对象（参见 `shelve`）。

```
$ python pickle_stream.py
```

```

WRITING : pickle (elkcip)
WRITING : cPickle (elkciPc)
WRITING : last (tsal)
READ    : pickle (elkcip)

```

```

READ      : cPickle (elkciPc)
READ      : last (tsal)

```

除了存储数据，pickle 对于进程间通信也很方便。例如，`os.fork()` 和 `os.pipe()` 可以用来建立工作进程，从一个管道读取作业指令，并把结果写至另一个管道。管理工作线程池以及发送作业和接收响应的核心代码可以重用，因为作业和响应对象不必基于一个特定的类。使用管道或套接字时，在转储各个对象之后不要忘记刷新输出，将数据通过连接推至另一端。要了解可重用的工作线程池管理器，可以参见 `multiprocessing` 模块。

7.1.4 重构对象的问题

处理定制类时，pickle 类必须出现在读取 pickle 的进程所在的命名空间。只会 pickle 这个实例的数据，而不包括类定义。类名用于查找构造函数，以便在解除 pickle 时创建新对象。下面这个例子将一个类的实例写至一个文件。

```

try:
    import cPickle as pickle
except:
    import pickle
import sys

class SimpleObject(object):
    def __init__(self, name):
        self.name = name
        l = list(name)
        l.reverse()
        self.name_backwards = ''.join(l)
        return

if __name__ == '__main__':
    data = []
    data.append(SimpleObject('pickle'))
    data.append(SimpleObject('cPickle'))
    data.append(SimpleObject('last'))

    filename = sys.argv[1]
    with open(filename, 'wb') as out_s:
        # Write to the stream
        for o in data:
            print 'WRITING: %s (%s)' % (o.name, o.name_backwards)
            pickle.dump(o, out_s)

```

运行这个脚本时，会根据作为命令行参数给定的名字创建一个文件。

```
$ python pickle_dump_to_file_1.py test.dat
```

```
WRITING: pickle (elkcip)
```

```

WRITING: cPickle (elkciPc)
WRITING: last (tsal)

```

如果简单地试图加载得到的 pickle 对象，将会失败。

```

try:
    import cPickle as pickle
except:
    import pickle
import pprint
from StringIO import StringIO
import sys

filename = sys.argv[1]

with open(filename, 'rb') as in_s:
    # Read the data
    while True:
        try:
            o = pickle.load(in_s)
        except EOFError:
            break
        else:
            print 'READ: %s (%s)' % (o.name, o.name_backwards)

```

这个版本失败的原因在于并没有 SimpleObject 类。

```

$ python pickle_load_from_file_1.py test.dat
Traceback (most recent call last):
  File "pickle_load_from_file_1.py", line 25, in <module>
    o = pickle.load(in_s)
AttributeError: 'module' object has no attribute 'SimpleObject'

```

修正后的版本从原脚本导入 SimpleObject，这一次运行会成功。在导入列表的最后添加以下 import 语句，从而允许脚本查找类并构造对象。

```

from pickle_dump_to_file_1 import SimpleObject

```

现在运行修改后的脚本会生成期望的结果。

```

$ python pickle_load_from_file_2.py test.dat

READ: pickle (elkcip)
READ: cPickle (elkciPc)
READ: last (tsal)

```

7.1.5 不可 pickle 的对象

并不是所有对象都是可 pickle 的。套接字、文件句柄、数据库连接以及其他运行时状态依赖于操作系统或其他进程的对象可能无法用一种有意义的方式保存。如果对象包含不可 pickle

的属性，可以定义 `__getstate__()` 和 `__setstate__()` 来返回可 pickle 实例状态的一个子集。新式的类还可以定义 `__getnewargs__()`，这会返回要传至类内存分配器 (`C.__new__()`) 的参数。这些特性的使用在标准库文档中有更详细的介绍。

7.1.6 循环引用

pickle 协议会自动处理对象之间的循环引用，所以复杂数据结构不需要任何特殊的处理。考虑图 7.1 中的有向图。图中包含几个循环，不过仍然可以 pickle 正确的结构然后重新加载。

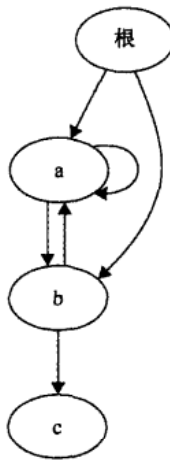


图 7.1 pickle 带循环的数据结构

```

import pickle

class Node(object):
    """A simple digraph"""
    def __init__(self, name):
        self.name = name
        self.connections = []

    def add_edge(self, node):
        "Create an edge between this node and the other."
        self.connections.append(node)

    def __iter__(self):
        return iter(self.connections)

def preorder_traversal(root, seen=None, parent=None):
    """Generator function to yield the edges in a graph.
    """
    if seen is None:
        seen = set()
    yield (parent, root)
  
```

```

    if root in seen:
        return
    seen.add(root)
    for node in root:
        for parent, subnode in preorder_traversal(node, seen, root):
            yield (parent, subnode)
def show_edges(root):
    "Print all the edges in the graph."
    for parent, child in preorder_traversal(root):
        if not parent:
            continue
        print '%5s -> %2s (%s)' % \
            (parent.name, child.name, id(child))

# Set up the nodes.
root = Node('root')
a = Node('a')
b = Node('b')
c = Node('c')

# Add edges between them.
root.add_edge(a)
root.add_edge(b)
a.add_edge(b)
b.add_edge(a)
b.add_edge(c)
a.add_edge(a)

print 'ORIGINAL GRAPH:'
show_edges(root)

# Pickle and unpickle the graph to create
# a new set of nodes.
dumped = pickle.dumps(root)
reloaded = pickle.loads(dumped)

print '\nRELOADED GRAPH:'
show_edges(reloaded)

```

重新加载的节点并不是同一个对象，不过节点之间的关系得到了维护，而且如果对象有多个引用，那么只会重新加载它的一个副本。要验证这两点，可以对通过 pickle 传递之前和传递之后的节点的 id() 值进行检查。

```
$ python pickle_cycle.py
```

```

ORIGINAL GRAPH:
root ->  a (4309376848)
a ->    b (4309376912)

```

```

b -> a (4309376848)
b -> c (4309376976)
a -> a (4309376848)
root -> b (4309376912)

```

RELOADED GRAPH:

```

root -> a (4309418128)
a -> b (4309418192)
b -> a (4309418128)
b -> c (4309418256)
a -> a (4309418128)
root -> b (4309418192)

```

参见:

`pickle` (<http://docs.python.org/lib/module-pickle.html>) 这个模块的标准库文档。

`Pickle: An interesting stack language` (<http://peadrop.com/blog/2007/06/18/pickle-an-interesting-stack-language/>) Alexandre Vassalotti 撰写的一篇博客帖子。

`Why Python Pickle is Insecure` (<http://nadiana.com/python-pickle-insecure>) 这是 Nadia Alramli 提供的一个简短例子, 展示了使用 `pickle` 存在的一个安全漏洞。

`shelve` (7.2 节) `shelve` 模块使用 `pickle` 在 DBM 数据库中存储数据。

7.2 shelve——对象持久存储

作用: `shelve` 模块使用一种类字典的 API, 可以持久存储可 `pickle` 的任意 Python 对象。

不需要关系数据库时, `shelve` 模块可以用作 Python 对象的一个简单的持久存储选择。类似于字典, `shelf` 要按键来访问。值将被 `pickle` 并写至由 `anydbm` 创建和管理的数据库。

7.2.1 创建一个新 shelf

使用 `shelve` 最简单的方法就是通过 `DbfilenameShelf` 类。它使用 `anydbm` 存储数据。这个类可以直接使用, 也可以通过调用 `shelve.open()` 来使用。

```

import shelve
from contextlib import closing
with closing(shelve.open('test_shelf.db')) as s:
    s['key1'] = { 'int': 10, 'float': 9.5, 'string': 'Sample data' }

```

要再次访问这个数据, 可以打开 `shelf`, 像字典一样使用。

```

import shelve
from contextlib import closing

with closing(shelve.open('test_shelf.db')) as s:
    existing = s['key1']

print existing

```

运行这两个示例脚本会产生以下结果。

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

dbm 模块不支持多个应用同时写同一个数据库，不过它支持并发的只读客户。如果一个客户没有修改 shelf，可以传入 flag='r' 告诉 shelve 以只读方式打开数据库。

```
import shelve
from contextlib import closing

with closing(shelve.open('test_shelf.db', flag='r')) as s:
    existing = s['key1']

print existing
```

如果数据库以只读模式打开，此时倘若一个程序试图修改这个数据库，会生成一个访问错误异常。具体的异常类型取决于创建数据库时 anydbm 选择的数据库模块。

7.2.2 写回

默认情况下，shelf 不会跟踪对可变对象的修改。这说明，如果存储在 shelf 中的一个元素内容有变化，shelf 必须通过再次存储整个元素来显式更新。

```
import shelve
from contextlib import closing

with closing(shelve.open('test_shelf.db')) as s:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'

with closing(shelve.open('test_shelf.db', writeback=True)) as s:
    print s['key1']
```

在这个例子中，'key1' 的相应字典没有再次存储，所以重新打开 shelf 时，修改不会保留。

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

对于 shelf 中存储的可变对象，为了自动捕获其修改，打开 shelf 时可以启用写回 (writeback)。写回标志使得 shelf 使用内存中缓存记住从数据库获取的所有对象。shelf 关闭时每个缓存对象也写回到数据库。

```

import shelve
import pprint
from contextlib import closing

with closing(shelve.open('test_shelf.db', writeback=True)) as s:
    print 'Initial data:'
    pprint.pprint(s['key1'])

    s['key1']['new_value'] = 'this was not here before'
    print '\nModified:'
    pprint.pprint(s['key1'])

with closing(shelve.open('test_shelf.db', writeback=True)) as s:
    print '\nPreserved:'
    pprint.pprint(s['key1'])

```

尽管这会减少程序员犯错的机会，并且能使对象持久存储更透明，但是并非所有情况下都有必要使用写回模式。打开 shelve 时缓存会消耗额外的内存，它关闭时会暂停将各个缓存对象写回到数据库，这会使应用的速度减慢。所有缓存的对象都要写回数据库，因为无法区分它们是否有修改。如果应用读取的数据多于写的数据，写回会影响性能而且没有太大意义。

```

$ python shelve_create.py
$ python shelve_writeback.py

Initial data:
{'float': 9.5, 'int': 10, 'string': 'Sample data'}

Modified:
{'float': 9.5,
 'int': 10,
 'new_value': 'this was not here before',
 'string': 'Sample data'}

Preserved:
{'float': 9.5,
 'int': 10,
 'new_value': 'this was not here before',
 'string': 'Sample data'}

```

7.2.3 特定 shelve 类型

之前的例子都使用了默认的 shelve 实现。可以使用 shelve.open() 而不是直接使用某种 shelve 实现，这是一种常见的用法，特别是当使用何种类型的数据库存储数据都无关紧要时。不过，有些情况下数据库格式会很重要。在这些情况下，可以直接使用 DbfilenameShelf 或 BsdDbShelf，或者甚至可以派生 Shelf 来得到一个定制的解决方案。

参见:

shelve (<http://docs.python.org/lib/module-shelve.html>) 这个模块的标准库文档。

feedcache (www.doughellmann.com/projects/feedcache/) feedcache 模块使用 shelve 作为默认的存储选项。

shove (<http://pypi.python.org/pypi/shove/>) Shove 实现了一个类似的 API, 不过提供了更多后端格式。

anydbm (7.3 节) anydbm 模块查找一个可用的 DBM 库来创建新的数据库。

7.3 anydbm——DBM 数据库

作用: anydbm 为以字符串为键的 DBM 数据库提供了一个通用的类字典接口。

Python 版本: 1.4 及以后版本

anydbm 是面向 DBM 数据库的一个前端, DBM 数据库使用简单的字符串值作为键来访问包含字符串的记录。anydbm 使用 whichdb 标识数据库, 然后用适当的模块打开这些数据库。它还用作为 shelve 的一个后端, shelve 使用 pickle 将对象存储在一个 DBM 数据库中。

7.3.1 数据库类型

Python 提供了很多模块来访问 DBM 数据库。具体选择哪个实现取决于当前系统上可用的库以及编译 Python 时使用的选项。

dbhash

dbhash 模块是 anydbm 的主要后端。它使用 bsddb 库来管理数据库文件。使用 dbhash 数据库的语义与 anydbm API 定义的语义相同。

gdbm

gdbm 是 GNU 项目 dbm 库的一个更新版本。其工作方式与这里介绍的其他 DBM 实现相同, 只是对 open() 支持的标志有些修改。

除了标准 'r'、'w'、'c' 和 'n' 标志, gdbm.open() 还支持以下标志:

- 'f' 以快速 (fast) 模式打开数据库。在快速模式下, 对数据库的写并不同步。
- 's' 以同步 (synchronized) 模式打开数据库。对数据库做出修改时, 这些改变要写至文件, 而不是延迟到数据库关闭或显式同步时才写至文件。
- 'u' 不加锁 (unlocked) 地打开数据库。

dbm

dbm 模块为 dbm 格式的某个 C 实现提供了一个接口, 具体哪个 C 实现取决于编译时如何配置这个模块。模块属性 library 指示编译扩展模块时 configure 能够找到的库名。

dumbdbm

dumbdbm 模块是没有其他实现可用时 DBM API 的一个可移植的后备实现。使用 dumbdbm

不要求依赖任何外部库，不过它比大多数其他实现速度都慢。

7.3.2 创建一个新数据库

会按顺序查找以下各个模块来选择新数据库的存储格式：

- dbhash
- gdbm
- dbm
- dumbdbm

open() 函数可以接收一些标志来控制如何管理数据库文件。必要时，要创建一个新的数据库，可以使用 'c'。使用 'n' 则总会创建一个新数据库而覆盖现有的文件。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'n')
db['key'] = 'value'
db['today'] = 'Sunday'
db['author'] = 'Doug'
db.close()
```

在这个例子中，文件总会重新初始化。

```
$ python anydbm_new.py
```

whichdb 会报告所创建的数据库的类型。

```
import whichdb

print whichdb.whichdb('/tmp/example.db')
```

取决于系统上安装的模块，示例程序的输出可能有所不同。

```
$ python anydbm_whichdb.py
```

```
dbhash
```

7.3.3 打开一个现有数据库

要打开一个现有数据库，可以使用标志 'r'（只读）或 'w'（读写）。会把现有的数据库自动提供给 whichdb 来识别，所以只要一个文件可以识别，就会使用一个适当的模块来打开这个文件。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'r')
try:
    print 'keys():', db.keys()
    for k, v in db.iteritems():
        print 'iterating:', k, v
```

```

    print 'db["author"] =', db['author']
finally:
    db.close()

```

一旦打开, db 是一个类字典的对象, 并支持所有常用的方法。

```
$ python anydbm_existing.py
```

```

keys(): ['author', 'key', 'today']
iterating: author Doug
iterating: key value
iterating: today Sunday
db["author"] = Doug

```

7.3.4 错误情况

数据库的键必须是字符串。

```

import anydbm

db = anydbm.open('/tmp/example.db', 'w')
try:
    db[1] = 'one'
except TypeError, err:
    print '%s: %s' % (err.__class__.__name__, err)
finally:
    db.close()

```

如果传入其他类型则会导致一个 `TypeError`。

```
$ python anydbm_intkeys.py
```

```
TypeError: Integer keys only allowed for Recno and Queue DB's
```

值必须是字符串或 `None`。

```

import anydbm

db = anydbm.open('/tmp/example.db', 'w')
try:
    db['one'] = 1
except TypeError, err:
    print '%s: %s' % (err.__class__.__name__, err)
finally:
    db.close()

```

如果值不是一个字符串, 也会产生一个类似的 `TypeError`。

```
$ python anydbm_intvalue.py
```

```
TypeError: Data values must be of type string or None.
```



参见:

anydbm (<http://docs.python.org/library/anydbm.html>) 这个模块的标准库文档。

shelve (7.2 节) shelve 模块的示例, 使用 anydbm 存储数据。

7.4 whichdb——识别 DBM 数据库格式

作用: 检查现有的 DBM 数据库文件, 以确定应当使用哪个库来打开该文件。

Python 版本: 1.4 及以后版本

whichdb 模块包含一个函数 whichdb(), 这个函数用来检查一个现有的数据库文件, 以确定应当使用哪个 DBM 库打开该文件。它会返回用来打开文件的模块的串名, 或者如果打开文件存在问题则返回 None。如果可以打开文件, 但是不能确定要使用的库, 则返回一个空串。

```
import anydbm
import whichdb

db = anydbm.open('/tmp/example.db', 'n')
db['key'] = 'value'
db.close()
```

```
print whichdb.whichdb('/tmp/example.db')
```

取决于系统上可用的模块, 运行这个示例程序的结果可能有所不同。

```
$ python whichdb_whichdb.py
```

```
dbhash
```

参见:

whichdb (<http://docs.python.org/lib/module-whichdb.html>) 这个模块的标准库文档。

anydbm (7.3 节) anydbm 模块创建新数据库时使用当前最好的 DBM 实现。

shelve (7.2 节) shelve 模块为 DBM 数据库提供了一个映射形式的 API。

7.5 sqlite3——嵌入式关系数据库

作用: 实现一个嵌入式关系数据库, 并提供 SQL 支持。

Python 版本: 2.5 及以后版本

sqlite3 模块为 SQLite 提供了一个 DB-API 2.0 兼容接口, SQLite 是一个进程中关系数据库。SQLite 设计为嵌入在应用中, 而不是像 MySQL、PostgreSQL 或 Oracle 使用一个单独的数据库服务器程序。SQLite 的速度很快、经过了严格的测试, 而且很灵活, 所以非常适合为一些应用建立原型和生产部署。

7.5.1 创建数据库

SQLite 数据库作为一个文件存储在文件系统中。这个库管理对文件的访问，包括加锁来防止多个书写器使用它时造成破坏。数据库在第一次访问文件时创建，不过应用要负责管理数据库中的数据库表定义，即模式（schema）。

下面这个例子在用 `connect()` 打开数据库文件之前先查找这个文件，以便了解何时为新数据库创建模式。

```
import os
import sqlite3

db_filename = 'todo.db'

db_is_new = not os.path.exists(db_filename)

conn = sqlite3.connect(db_filename)

if db_is_new:
    print 'Need to create schema'
else:
    print 'Database exists, assume schema does, too.'

conn.close()
```

将这个脚本运行两次，可以看到，如果文件尚不存在则会创建空文件。

```
$ ls *.db

ls: *.db: No such file or directory

$ python sqlite3_createdb.py

Need to create schema

$ ls *.db

todo.db

$ python sqlite3_createdb.py

Database exists, assume schema does, too.
```

创建新的数据库文件后，下一步是创建模式来定义数据库中的表。这一节余下的例子使用的数据库模式与管理任务的表相同。数据库模式的详细信息见表 7.1 和表 7.2。

表 7.1 “project” 表

列	类 型	描 述
name	text	项目名
description	text	详细的项目描述
deadline	date	整个项目的预定结束日期

表 7.2 “task” 表

列	类 型	描 述
Id	number	惟一任务标识符
priority	integer	优先级数值；值越小越重要
details	text	完备的任务详细描述
status	text	任务状态 [可以是 new (新建)、pending (未完成)、done (完成) 或 canceled (取消) 之一]
Deadline	date	这个任务的预定结束日期
completed_on	date	任务何时完成
project	text	这个任务对应的项目名

以下是创建这些表的数据定义语言 (data definition language , DDL) 语句。

```
-- Schema for to-do application examples.

-- Projects are high-level activities made up of tasks
create table project (
    name      text primary key,
    description text,
    deadline  date
);

-- Tasks are steps that can be taken to complete a project
create table task (
    id          integer primary key autoincrement not null,
    priority    integer default 1,
    details     text,
    status      text,
    deadline    date,
    completed_on date,
    project     text not null references project(name)
);
```

可以用 Connection 的 `executescript()` 方法来运行创建模式的 DDL 指令。

```
import os
import sqlite3
```

```

db_filename = 'todo.db'
schema_filename = 'todo_schema.sql'

db_is_new = not os.path.exists(db_filename)

with sqlite3.connect(db_filename) as conn:
    if db_is_new:
        print 'Creating schema'
        with open(schema_filename, 'rt') as f:
            schema = f.read()
            conn.executescript(schema)

        print 'Inserting initial data'

        conn.executescript("""
insert into project (name, description, deadline)
values ('pymotw', 'Python Module of the Week', '2010-11-01');

insert into task (details, status, deadline, project)
values ('write about select', 'done', '2010-10-03',
        'pymotw');

insert into task (details, status, deadline, project)
values ('write about random', 'waiting', '2010-10-10',
        'pymotw');

insert into task (details, status, deadline, project)
values ('write about sqlite3', 'active', '2010-10-17',
        'pymotw');
""")
    else:
        print 'Database exists, assume schema does, too.'

```

创建这些数据表之后，用一些插入语句创建一个示例项目和相关的任务。可以用 `sqlite3` 命令行程序检查数据库的内容。

```

$ python sqlite3_create_schema.py

Creating schema
Inserting initial data

$ sqlite3 todo.db 'select * from task'

1|1|write about select|done|2010-10-03||pymotw
2|1|write about random|waiting|2010-10-10||pymotw
3|1|write about sqlite3|active|2010-10-17||pymotw

```

7.5.2 获取数据

要从一个 Python 程序中获取 task 表中保存的值，可以从数据库连接创建一个 cursor。游标 (cursor) 会生成一个一致的数据视图，这也是与类似 SQLite 的事务型数据库系统交互的主要方式。

```
import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
        select id, priority, details, status, deadline from task
        where project = 'pymotw'
        """)

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print '%2d (%d) %-20s [%-8s] (%s)' % \
            (task_id, priority, details, status, deadline)
```

查询过程包括两步。首先，用游标的 `execute()` 方法运行查询，告诉数据库引擎要收集哪些数据。然后，使用 `fetchall()` 获取结果。返回值是一个元组序列，元组中包含查询 `select` 子句中所包括的列的值。

```
$ python sqlite3_select_tasks.py

1 {1} write about select    [done    ] (2010-10-03)
2 {1} write about random   [waiting] (2010-10-10)
3 {1} write about sqlite3  [active ] (2010-10-17)
```

可以用 `fetchone()` 一次获取一个结果，也可以用 `fetchmany()` 获取固定大小的批量结果。

```
import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
        select name, description, deadline from project
        where name = 'pymotw'
        """)
    name, description, deadline = cursor.fetchone()
```

```

print 'Project details for %s (%s) due %s' % \
      (description, name, deadline)

cursor.execute("""
select id, priority, details, status, deadline from task
where project = 'pymotw' order by deadline
""")

print '\nNext 5 tasks:'
for row in cursor.fetchmany(5):
    task_id, priority, details, status, deadline = row
    print '%2d %d %s %-25s [%s] (%s)' % \
          (task_id, priority, details, status, deadline)

```

传入 `fetchmany()` 的值是要返回的最大元素数。如果没有提供足够的元素，返回的序列大小将小于这个最大值。

```
$ python sqlite3_select_variations.py
```

```
Project details for Python Module of the Week (pymotw) due 2010-11-01
```

```
Next 5 tasks:
```

```

1 {1} write about select      [done    ] (2010-10-03)
2 {1} write about random     [waiting ] (2010-10-10)
3 {1} write about sqlite3    [active  ] (2010-10-17)

```

7.5.3 查询元数据

DB-API 2.0 规范指出：调用 `execute()` 之后，`cursor` 应当设置其 `description` 属性，来保存将由 `fetch` 方法返回的数据的有关信息。API 规范指出这个描述值是一个元组序列，各元组包含列名、类型、显示大小、内部大小、精度、范围和一个指示是否接受 `null` 值的标志。

```

import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
select * from task where project = 'pymotw'
""")

    print 'Task table has these columns:'
    for colinfo in cursor.description:
        print colinfo

```

由于 `sqlite3` 对插入到数据库的数据没有类型或大小约束，所以只填入列名值。



```
$ python sqlite3_cursor_description.py
```

```
Task table has these columns:
('id', None, None, None, None, None, None)
('priority', None, None, None, None, None, None)
('details', None, None, None, None, None, None)
('status', None, None, None, None, None, None)
('deadline', None, None, None, None, None, None)
('completed_on', None, None, None, None, None, None).
('project', None, None, None, None, None, None)
```

7.5.4 行对象

默认情况下，获取方法从数据库作为“行”返回的值是元组。调用者负责了解查询中列的顺序，并从元组中抽取单个的值。查询的值个数增加时，或者处理数据的代码分布在一个库的不同位置时，通常更容易的做法是处理一个对象，并使用其列名来访问值。这样一来，编辑查询时，元组内容的个数和顺序可以随时间改变，另外依赖于查询结果的代码也不太可能出问题。

Connection 对象有一个 row_factory 属性，允许调用代码控制所创建对象的类型来表示查询结果集中的各行。sqlite3 还包括一个 Row 类，这个类将用作一个行工厂。可以通过 Row 实例使用列索引或名来访问列值。

```
import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    # Change the row factory to use Row
    conn.row_factory = sqlite3.Row

    cursor = conn.cursor()

    cursor.execute("""
select name, description, deadline from project
where name = 'pymotw'
""")
    name, description, deadline = cursor.fetchone()

    print 'Project details for %s (%s) due %s' % (
        description, name, deadline)

    cursor.execute("""
select id, priority, status, deadline, details from task
where project = 'pymotw' order by deadline
""")
```

```

print '\nNext 5 tasks:'
for row in cursor.fetchmany(5):
    print '%2d (%d) %-25s [%-8s] (%s)' % (
        row['id'], row['priority'], row['details'],
        row['status'], row['deadline'],
    )

```

这个版本的 `sqlite3_select_variations.py` 例子重写为使用 `Row` 实例而不是元组。打印 `project` 表中的行时仍然通过位置来访问列值，不过打印任务的 `print` 语句使用了关键字查找，所以任务查询中列顺序的改变不会有任何影响。

```
$ python sqlite3_row_factory.py
```

```
Project details for Python Module of the Week (pymotw) due 2010-11-01
```

```
Next 5 tasks:
```

```

1 {1} write about select      [done    ] (2010-10-03)
2 {1} write about random     [waiting ] (2010-10-10)
3 {1} write about sqlite3    [active  ] (2010-10-17)

```

7.5.5 查询中使用变量

如果查询定义为字面量字符串嵌入到程序中，使用这种查询很不灵活。例如，向数据库添加另一个项目时，显示前 5 个任务的查询就应当更新，以处理其中某一个项目。要想增加灵活性，一种方法是建立一个 SQL 语句，通过在 Python 中结合相应的值来得到所需的查询。不过，以这种方式构造查询串很危险，应当尽量避免。如果未能对查询中可变部分的特殊字符正确转义，可能会导致 SQL 解析错误，或者更糟糕的是，还有可能导致一个安全漏洞，称为 SQL 注入攻击 (SQL-injection attack)，这使得入侵者可以在数据库中执行任意的 SQL 语句。

要在查询中使用动态值，正确的方法是利用随 SQL 指令一起传入 `execute()` 的宿主变量 (host variable)。SQL 语句执行时，语句中的占位符值会替换为宿主变量的值。通过使用宿主变量，而不是解析之前在 SQL 语句中插入任意的值，这样可以避免注入攻击，因为不可信的值没有机会影响 SQL 语句的解析。SQLite 支持两种形式带占位符的查询，分别是位置参数和命名参数。

位置参数

问号 (?) 指示一个位置参数，将作为元组的一个成员传至 `execute()`。

```

import sqlite3
import sys

```

```

db_filename = 'todo.db'
project_name = sys.argv[1]

```

```

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

```



```

query = """select id, priority, details, status, deadline from task
          where project = ?
          """

cursor.execute(query, (project_name,))

for row in cursor.fetchall():
    task_id, priority, details, status, deadline = row
    print '%2d (%d) %-20s [%-8s] (%s)' % (
        task_id, priority, details, status, deadline)

```

命令行参数会作为位置参数安全地传至查询，所以恶意数据不可能破坏数据库。

```
$ python sqlite3_argument_positional.py pymotw
```

```

1 {1} write about select      [done      ] (2010-10-03)
2 {1} write about random     [waiting ] (2010-10-10)
3 {1} write about sqlite3    [active  ] (2010-10-17)

```

命名参数

对于包含大量参数的更为复杂的查询，或者如果查询中某些参数会重复多次，则可以使用命名参数。命名参数前面有一个冒号作为前缀（例如，:param_name）。

```

import sqlite3
import sys
db_filename = 'todo.db'
project_name = sys.argv[1]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    query = """select id, priority, details, status, deadline from task
              where project = :project_name
              order by deadline, priority
              """

    cursor.execute(query, {'project_name':project_name})

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print '%2d (%d) %-25s [%-8s] (%s)' % (\
            task_id, priority, details, status, deadline)

```

位置或命名参数都不需要加引号或转义，因为查询解析器会对它们做特殊处理。

```
$ python sqlite3_argument_named.py pymotw
```

```
1 {1} write about select      [done      ] (2010-10-03)
```

```

2 {1} write about random      [waiting ] (2010-10-10)
3 {1} write about sqlite3     [active  ] (2010-10-17)

```

查询参数可以在 `select` (选择)、`insert` (插入) 和 `update` (更新) 语句中使用。查询中字面量能够出现的位置都可以放置查询参数。

```

import sqlite3
import sys

db_filename = 'todo.db'
id = int(sys.argv[1])
status = sys.argv[2]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
    query = "update task set status = :status where id = :id"
    cursor.execute(query, {'status':status, 'id':id})

```

这个 `update` 语句使用了两个命名参数。id 值用于查找要修改的行，status 值则要写入数据表。

```

$ python sqlite3_argument_update.py 2 done
$ python sqlite3_argument_named.py pymotw

```

```

1 {1} write about select      [done    ] (2010-10-03)
2 {1} write about random      [done    ] (2010-10-10)
3 {1} write about sqlite3     [active  ] (2010-10-17)

```

7.5.6 批量加载

要对一个很大的数据集应用相同的 SQL 指令，可以使用 `executemany()`。这对于加载数据很有用，因为这样可以避免在 Python 中循环处理输入，而是让底层库对循环应用一些优化。下面这个示例程序使用 `csv` 模块从一个逗号分隔值文件读取任务列表，并将其加载到数据库。

```

import csv
import sqlite3
import sys

db_filename = 'todo.db'
data_filename = sys.argv[1]

SQL = """
    insert into task (details, priority, status, deadline, project)
    values (:details, :priority, 'active', :deadline, :project)
    """

with open(data_filename, 'rt') as csv_file:
    csv_reader = csv.DictReader(csv_file)

    with sqlite3.connect(db_filename) as conn:

```

```
cursor = conn.cursor()
cursor.executemany(SQL, csv_reader)
```

示例数据文件 `tasks.csv` 包含以下数据:

```
deadline,project,priority,details
2010-10-02,pymotw,2,"finish reviewing markup"
2010-10-03,pymotw,2,"revise chapter intros"
2010-10-03,pymotw,1,"subtitle"
```

运行这个程序会生成以下结果:

```
$ python sqlite3_load_csv.py tasks.csv
$ python sqlite3_argument_named.py pymotw
```

```
4 {2} finish reviewing markup [active ] (2010-10-02)
1 {1} write about select      [done  ] (2010-10-03)
6 {1} subtitle                [active ] (2010-10-03)
5 {2} revise chapter intros   [active ] (2010-10-03)
2 {1} write about random      [done  ] (2010-10-10)
3 {1} write about sqlite3     [active ] (2010-10-17)
```

7.5.7 定义新列类型

SQLite 对整数、浮点数和文本列提供了内置支持。sqlite3 会自动将这些类型的数据从 Python 的表示转换为可在数据库中存储的一个值，还可以根据需要从数据库中存储的值转换回 Python 的表示。整数值由数据库加载为 `int` 或 `long` 变量，这取决于值的大小。文本将作为 `unicode` 保存和获取（除非改变了 `Connection` 的 `text_factory`）。

尽管 SQLite 在内部只支持几种数据类型，不过 sqlite3 包括了一些便利工具，可以定义定制类型，允许 Python 应用在列中存储任意类型的数据。除了那些得到默认支持的类型外，还可以在数据库连接中使用 `detect_types` 标志启用其他类型的转换。如果定义表时列使用所要求的类型来声明，可以使用 `PARSE_DECLTYPES`。

```
import sqlite3
import sys

db_filename = 'todo.db'

sql = "select id, details, deadline from task"

def show_deadline(conn):
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()
    cursor.execute(sql)
    row = cursor.fetchone()
    for col in ['id', 'details', 'deadline']:
        print ' %-8s %-30s %s' % (col, row[col], type(row[col]))
    return
```



```

print 'Without type detection:'
with sqlite3.connect(db_filename) as conn:
    show_deadline(conn)

print '\nWith type detection:'
with sqlite3.connect(db_filename,
                    detect_types=sqlite3.PARSE_DECLTYPES,
                    ) as conn:
    show_deadline(conn)

```

sqlite3 为日期和时间戳列提供了转换器，它使用 datetime 模块的 date 和 datetime 表示 Python 中的值。这两个与日期有关的转换器会在打开类型检测时自动启用。

```
$ python sqlite3_date_types.py
```

```

Without type detection:
id          1                                <type 'int'>
details     u'write about select'           <type 'unicode'>
deadline    u'2010-10-03'                   <type 'unicode'>

With type detection:
id          1                                <type 'int'>
details     u'write about select'           <type 'unicode'>
deadline    datetime.date(2010, 10, 3)      <type 'datetime.date'>

```

定义一个新类型需要注册两个函数。适配器 (adapter) 取 Python 对象作为输入，返回一个可以存储在数据库中的字节串。转换器 (converter) 从数据库接收串，返回一个 Python 对象。要使用 register_adapter() 定义适配器函数，使用 register_converter() 定义转换器函数。

```

import sqlite3
try:
    import cPickle as pickle
except:
    import pickle
db_filename = 'todo.db'

def adapter_func(obj):
    """Convert from in-memory to storage representation.
    """
    print 'adapter_func(%s)\n' % obj
    return pickle.dumps(obj)

def converter_func(data):
    """Convert from storage to in-memory representation.
    """
    print 'converter_func(%r)\n' % data
    return pickle.loads(data)

```

```

class MyObj(object):
    def __init__(self, arg):
        self.arg = arg
    def __str__(self):
        return 'MyObj(%r)' % self.arg

# Register the functions for manipulating the type.
sqlite3.register_adapter(MyObj, adapter_func)
sqlite3.register_converter("MyObj", converter_func)

# Create some objects to save. Use a list of tuples so
# the sequence can be passed directly to executemany().
to_save = [ (MyObj('this is a value to save'),),
             (MyObj(42),),
             ]

with sqlite3.connect(db_filename,
                    detect_types=sqlite3.PARSE_DECLTYPES) as conn:
    # Create a table with column of type "MyObj"
    conn.execute("""
create table if not exists obj (
    id      integer primary key autoincrement not null,
    data    MyObj
)
""")
    cursor = conn.cursor()

    # Insert the objects into the database
    cursor.executemany("insert into obj (data) values (?)", to_save)
    # Query the database for the objects just saved
    cursor.execute("select id, data from obj")
    for obj_id, obj in cursor.fetchall():
        print 'Retrieved', obj_id, obj, type(obj)
    print

```

这个例子使用 `pickle` 将一个对象保存为可以存储在数据库中的串，这对于存储任意的对象很有用，不过这种技术不支持按对象属性查询。真正的对象关系映射器（object-relational mapper，如 `SQLAlchemy`）可以将属性值存储在单独的列中，这对于大量数据更为有用。

```
$ python sqlite3_custom_type.py
```

```
adapter_func(MyObj('this is a value to save'))
```

```
adapter_func(MyObj(42))
```

```
converter_func("ccopy_reg\n_reconstructor\npl\n(c__main__\nMyObj\np2
```

```
\nc__builtin__\nobject\np3\nNtRp4\n(dp5\ns'arg'\np6\ns'this is a value to save'\np7\nsb.)
```

```
converter_func("ccopy_reg\n_reconstructor\np1\n(c__main__\nMyObj\np2\nnc__builtin__\nobject\np3\nNtRp4\n(dp5\ns'arg'\np6\nI42\nsb.)")
```

```
Retrieved 1 MyObj('this is a value to save') <class '__main__.MyObj'>
```

```
Retrieved 2 MyObj(42) <class '__main__.MyObj'>
```

7.5.8 确定列类型

查询返回的值的类型信息有两个来源。可以用原表声明来识别一个实际列的类型，这在前面已经看到。另外还可以在查询自身的 `select` 子句中包含类型指示符，采用以下形式：`as "name [type]"`。

```
import sqlite3
try:
    import cPickle as pickle
except:
    import pickle
db_filename = 'todo.db'

def adapter_func(obj):
    """Convert from in-memory to storage representation.
    """
    print 'adapter_func(%s)\n' % obj
    return pickle.dumps(obj)

def converter_func(data):
    """Convert from storage to in-memory representation.
    """
    print 'converter_func(%r)\n' % data
    return pickle.loads(data)

class MyObj(object):
    def __init__(self, arg):
        self.arg = arg
    def __str__(self):
        return 'MyObj(%r)' % self.arg

# Register the functions for manipulating the type.
sqlite3.register_adapter(MyObj, adapter_func)
sqlite3.register_converter("MyObj", converter_func)
```

```

# Create some objects to save. Use a list of tuples so we can pass
# this sequence directly to executemany().
to_save = [ (MyObj('this is a value to save'),),
             (MyObj(42),),
             ]

with sqlite3.connect(db_filename,
                     detect_types=sqlite3.PARSE_COLNAMES) as conn:
    # Create a table with column of type "text"
    conn.execute("""
create table if not exists obj2 (
    id    integer primary key autoincrement not null,
    data  text
)
""")
    cursor = conn.cursor()

    # Insert the objects into the database
    cursor.executemany("insert into obj2 (data) values (?)", to_save)
    # Query the database for the objects just saved,
    # using a type specifier to convert the text
    # to objects.
    cursor.execute('select id, data as "pickle [MyObj]" from obj2')
    for obj_id, obj in cursor.fetchall():
        print 'Retrieved', obj_id, obj, type(obj)
        print

```

如果类型是查询的一部分而不属于原表定义，则要使用 `detect_types` 标志 `PARSE_COLNAMES`。

```
$ python sqlite3_custom_type_column.py
```

```
adapter_func(MyObj('this is a value to save'))
```

```
adapter_func(MyObj(42))
```

```
converter_func("ccopy_reg\n_reconstructor\np1\n(c__main__\nMyObj\np2\nnc__builtin__\nobject\np3\nNtRp4\n(dp5\nS'arg'\np6\nS'this is a value to save'\np7\nsb.")
```

```
converter_func("ccopy_reg\n_reconstructor\np1\n(c__main__\nMyObj\np2\nnc__builtin__\nobject\np3\nNtRp4\n(dp5\nS'arg'\np6\nI42\nsb.")
```

```
Retrieved 1 MyObj('this is a value to save') <class '__main__.MyObj'>
```

```
Retrieved 2 MyObj(42) <class '__main__.MyObj'>
```

7.5.9 事务

关系数据库的关键特性之一是使用事务 (transaction) 维护一致的内部状态。启用事务时，在提交结果并刷新输出到真正的数据库之前，可以通过一个连接完成多个变更，而不会影响任何其他用户。

保留变更

不论通过插入 (insert) 还是更新 (update) 语句改变数据库，都需要显式地调用 `commit()` 保存这些变更。这个要求为应用提供了一个机会，可以将多个相关的变更一同完成，使它们以一种“原子”方式保存而不是增量保存，这样可以避免同时连接到数据库的不同客户只看到部分更新的情况。

可以利用一个使用了多个数据库连接的程序来查看调用 `commit()` 的效果。用第一个连接插入一个新行，然后两次尝试使用不同的连接读回这个数据行。

```
import sqlite3

db_filename = 'todo.db'

def show_projects(conn):
    cursor = conn.cursor()
    cursor.execute('select name, description from project')
    for name, desc in cursor.fetchall():
        print ' ', name
    return

with sqlite3.connect(db_filename) as conn1:

    print 'Before changes:'
    show_projects(conn1)

    # Insert in one cursor
    cursor1 = conn1.cursor()
    cursor1.execute("""
insert into project (name, description, deadline)
values ('virtualenvwrapper', 'Virtualenv Extensions',
        '2011-01-01')
""")

    print '\nAfter changes in conn1:'
    show_projects(conn1)

    # Select from another connection, without committing first
    print '\nBefore commit:'
    with sqlite3.connect(db_filename) as conn2:
        show_projects(conn2)
```



```

# Commit then select from another connection
conn1.commit()
print '\nAfter commit:'
with sqlite3.connect(db_filename) as conn3:
    show_projects(conn3)

```

提交 conn1 之前调用 show_projects() 时，其结果取决于使用了哪个连接。由于这个改变通过 conn1 完成，它会看到修改后的数据。不过，conn2 看不到这个改变。提交之后，新连接 conn3 会看到插入的行。

```
$ python sqlite3_transaction_commit.py
```

```
Before changes:
pymotw
```

```
After changes in conn1:
pymotw
virtualenvwrapper
```

```
Before commit:
pymotw
```

```
After commit:
pymotw
virtualenvwrapper
```

丢弃变更

还可以使用 rollback() 完全丢弃未提交的变更。commit() 和 rollback() 方法通常在同一个 try:except 块的不同部分调用，有错误时就会触发回滚。

```

import sqlite3

db_filename = 'todo.db'

def show_projects(conn):
    cursor = conn.cursor()
    cursor.execute('select name, description from project')
    for name, desc in cursor.fetchall():
        print ' ', name
    return

with sqlite3.connect(db_filename) as conn:

    print 'Before changes:'
    show_projects(conn)
    try:

```



```

    # Insert
    cursor = conn.cursor()
    cursor.execute("""delete from project
                    where name = 'virtualenvwrapper'
                    """)

    # Show the settings
    print '\nAfter delete:'
    show_projects(conn)

    # Pretend the processing caused an error
    raise RuntimeError('simulated error')

except Exception, err:
    # Discard the changes
    print 'ERROR:', err
    conn.rollback()

else:
    # Save the changes
    conn.commit()

    # Show the results
    print '\nAfter rollback:'
    show_projects(conn)

```

调用 `rollback()` 后，对数据库的修改不复存在。

```
$ python sqlite3_transaction_rollback.py
```

Before changes:

```
pymotw
virtualenvwrapper
```

After delete:

```
pymotw
ERROR: simulated error
```

After rollback:

```
pymotw
virtualenvwrapper
```

7.5.10 隔离级别

sqlite3 支持 3 种加锁模式，也称为隔离级别 (isolation level)，这会控制使用何种技术避免连接之间不兼容的变更。打开一个连接时可以传入一个字符串作为 `isolation_level` 参数来设置隔离级别，所以不同的连接可以使用不同的隔离级别值。

下面这个程序展示了使用同一个数据库的不同连接时，不同的隔离级别对于线程中事件的顺序会有什么影响。这里创建了4个线程。两个线程会更新现有的行，将变更写入数据库。另外两个线程尝试从 task 表读取所有行。

```
import logging
import sqlite3
import sys
import threading
import time

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s %(threadName)-10s %(message)s',
)

db_filename = 'todo.db'
isolation_level = sys.argv[1]

def writer():
    my_name = threading.currentThread().name
    with sqlite3.connect(db_filename,
                        isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        cursor.execute('update task set priority = priority + 1')
        logging.debug('waiting to synchronize')
        ready.wait() # synchronize threads
        logging.debug('PAUSING')
        time.sleep(1)
        conn.commit()
        logging.debug('CHANGES COMMITTED')
    return

def reader():
    my_name = threading.currentThread().name
    with sqlite3.connect(db_filename,
                        isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        logging.debug('waiting to synchronize')
        ready.wait() # synchronize threads
        logging.debug('wait over')
        cursor.execute('select * from task')
        logging.debug('SELECT EXECUTED')
        results = cursor.fetchall()
        logging.debug('results fetched')
    return

if __name__ == '__main__':
```

```

ready = threading.Event()

threads = [
    threading.Thread(name='Reader 1', target=reader),
    threading.Thread(name='Reader 2', target=reader),
    threading.Thread(name='Writer 1', target=writer),
    threading.Thread(name='Writer 2', target=writer),
]

[ t.start() for t in threads ]

time.sleep(1)
logging.debug('setting ready')
ready.set()

[ t.join() for t in threads ]

```

这些线程使用 `threading` 模块的一个 `Event` 完成同步。`writer()` 函数连接数据库，并完成数据库修改，不过在事件触发前并不提交。`reader()` 函数连接数据库，然后等待查询数据库，直到出现同步事件。

延迟

默认的隔离级别是 `DEFERRED`。使用延迟 (Deferred) 模式会锁定数据库，但只是在修改真正开始时锁定一次。前面的所有例子都使用了延迟模式。

```
$ python sqlite3_isolation_levels.py DEFERRED
```

```

2010-12-04 09:06:51,793 (Reader 1 ) waiting to synchronize
2010-12-04 09:06:51,794 (Reader 2 ) waiting to synchronize
2010-12-04 09:06:51,795 (Writer 1 ) waiting to synchronize
2010-12-04 09:06:52,796 (MainThread) setting ready
2010-12-04 09:06:52,797 (Writer 1 ) PAUSING
2010-12-04 09:06:52,797 (Reader 1 ) wait over
2010-12-04 09:06:52,798 (Reader 1 ) SELECT EXECUTED
2010-12-04 09:06:52,798 (Reader 1 ) results fetched
2010-12-04 09:06:52,799 (Reader 2 ) wait over
2010-12-04 09:06:52,800 (Reader 2 ) SELECT EXECUTED
2010-12-04 09:06:52,800 (Reader 2 ) results fetched
2010-12-04 09:06:53,799 (Writer 1 ) CHANGES COMMITTED
2010-12-04 09:06:53,829 (Writer 2 ) waiting to synchronize
2010-12-04 09:06:53,829 (Writer 2 ) PAUSING
2010-12-04 09:06:54,832 (Writer 2 ) CHANGES COMMITTED

```

立即

采用立即 (Immediate) 模式时，修改一开始就会锁定数据库，从而在事务提交之前避免其他游标修改数据库。如果数据库有复杂的写操作，但是阅读器比书写器更多，这种模式就非

常适合，因为事务进行中不会阻塞阅读器。

```
$ python sqlite3_isolation_levels.py IMMEDIATE

2010-12-04 09:06:54,914 (Reader 1 ) waiting to synchronize
2010-12-04 09:06:54,915 (Reader 2 ) waiting to synchronize
2010-12-04 09:06:54,916 (Writer 1 ) waiting to synchronize
2010-12-04 09:06:55,917 (MainThread) setting ready
2010-12-04 09:06:55,918 (Reader 1 ) wait over
2010-12-04 09:06:55,919 (Reader 2 ) wait over
2010-12-04 09:06:55,919 (Writer 1 ) PAUSING
2010-12-04 09:06:55,919 (Reader 1 ) SELECT EXECUTED
2010-12-04 09:06:55,919 (Reader 1 ) results fetched
2010-12-04 09:06:55,920 (Reader 2 ) SELECT EXECUTED
2010-12-04 09:06:55,920 (Reader 2 ) results fetched
2010-12-04 09:06:56,922 (Writer 1 ) CHANGES COMMITTED
2010-12-04 09:06:56,951 (Writer 2 ) waiting to synchronize
2010-12-04 09:06:56,951 (Writer 2 ) PAUSING
2010-12-04 09:06:57,953 (Writer 2 ) CHANGES COMMITTED
```

互斥

互斥 (Exclusive) 模式会对所有阅读器和书写器锁定数据库。如果数据库性能很重要，这种情况下就要限制使用这种模式，因为每个互斥的连接都会阻塞所有其他用户。

```
$ python sqlite3_isolation_levels.py EXCLUSIVE

2010-12-04 09:06:58,042 (Reader 1 ) waiting to synchronize
2010-12-04 09:06:58,043 (Reader 2 ) waiting to synchronize
2010-12-04 09:06:58,044 (Writer 1 ) waiting to synchronize
2010-12-04 09:06:59,045 (MainThread) setting ready
2010-12-04 09:06:59,045 (Writer 1 ) PAUSING
2010-12-04 09:06:59,046 (Reader 2 ) wait over
2010-12-04 09:06:59,045 (Reader 1 ) wait over
2010-12-04 09:07:00,048 (Writer 1 ) CHANGES COMMITTED
2010-12-04 09:07:00,076 (Reader 1 ) SELECT EXECUTED
2010-12-04 09:07:00,076 (Reader 1 ) results fetched
2010-12-04 09:07:00,079 (Reader 2 ) SELECT EXECUTED
2010-12-04 09:07:00,079 (Reader 2 ) results fetched
2010-12-04 09:07:00,090 (Writer 2 ) waiting to synchronize
2010-12-04 09:07:00,090 (Writer 2 ) PAUSING
2010-12-04 09:07:01,093 (Writer 2 ) CHANGES COMMITTED
```

由于第一个书写器已经开始修改，所以阅读器和第二个书写器会阻塞，直到第一个书写器提交。`sleep()` 调用在书写器线程中引入一个人为的延迟，以强调其他连接已阻塞这一事实。

自动提交

连接的 `isolation_level` 参数还可以设置为 `None`，以启用自动提交 (autocommit) 模式。启

用自动提交时，每个 `execute()` 调用会在语句完成时立即提交。自动提交模式很适合简短的事务，如向一个表插入少量数据。数据库锁定时间尽可能短，所以线程间竞争的可能性更小。

`sqlite3_autocommit.py` 中删除了 `commit()` 的显式调用，并将隔离级别设置为 `None`，不过除此以外，其他内容都与 `sqlite3_isolation_levels.py` 相同。但输出是不同的，因为两个书写器线程会在阅读器开始查询之前完成工作。

```
$ python sqlite3_autocommit.py
```

```
2010-12-04 09:07:01,176 (Reader 1 ) waiting to synchronize
2010-12-04 09:07:01,177 (Reader 2 ) waiting to synchronize
2010-12-04 09:07:01,181 (Writer 1 ) waiting to synchronize
2010-12-04 09:07:01,184 (Writer 2 ) waiting to synchronize
2010-12-04 09:07:02,180 (MainThread) setting ready
2010-12-04 09:07:02,181 (Writer 1 ) PAUSING
2010-12-04 09:07:02,181 (Reader 1 ) wait over
2010-12-04 09:07:02,182 (Reader 1 ) SELECT EXECUTED
2010-12-04 09:07:02,182 (Reader 1 ) results fetched
2010-12-04 09:07:02,183 (Reader 2 ) wait over
2010-12-04 09:07:02,183 (Reader 2 ) SELECT EXECUTED
2010-12-04 09:07:02,184 (Reader 2 ) results fetched
2010-12-04 09:07:02,184 (Writer 2 ) PAUSING
```

7.5.11 内存中数据库

SQLite 支持在 RAM 中管理整个数据库，而不是依赖一个磁盘文件。如果测试运行之间不需要保留数据库，或者要尝试一个模式或其他数据库特性，此时内存中数据库对于自动测试会很有用。要打开一个内存中数据库，创建 `Connection` 时可以使用串 `:memory:` 而不是一个文件名。每个 `:memory:` 连接会创建一个单独的数据库实例，所以一个连接中游标所做的修改不会影响其他连接。

7.5.12 导出数据库内容

内存中数据库的内容可以使用 `Connection` 的 `iterdump()` 方法保存。`iterdump()` 方法返回的迭代器生成一系列字符串，这些字符串将共同构造相应的 SQL 指令来重新创建数据库的状态。

```
import sqlite3

schema_filename = 'todo_schema.sql'

with sqlite3.connect(':memory:') as conn:
    conn.row_factory = sqlite3.Row

    print 'Creating schema'
    with open(schema_filename, 'rt') as f:
        schema = f.read()
    conn.executescript(schema)
```

```

print 'Inserting initial data'
conn.execute("""
    insert into project (name, description, deadline)
    values ('pymotw', 'Python Module of the Week', '2010-11-01')
    """)
data = [
    ('write about select', 'done', '2010-10-03', 'pymotw'),
    ('write about random', 'waiting', '2010-10-10', 'pymotw'),
    ('write about sqlite3', 'active', '2010-10-17', 'pymotw'),
]
conn.executemany("""
    insert into task (details, status, deadline, project)
    values (?, ?, ?, ?)
    """, data)

print 'Dumping:'
for text in conn.iterdump():
    print text

```

iterdump() 也适用于保存到文件的数据库, 不过对于未保存的数据库最为有用。这里对输出做了一些编辑调整, 从而使其在保证语法正确的前提下适合在页面中显示。

```
$ python sqlite3_iterdump.py
```

```

Creating schema
Inserting initial data
Dumping:
BEGIN TRANSACTION;
CREATE TABLE project (
    name          text primary key,
    description text,
    deadline      date
);
INSERT INTO "project" VALUES('pymotw','Python Module of the
Week','2010-11-01');
CREATE TABLE task (
    id             integer primary key autoincrement not null,
    priority        integer default 1,
    details         text,
    status          text,
    deadline        date,
    completed_on date,
    project         text not null references project(name)
);
INSERT INTO "task" VALUES(1,1,'write about
select','done','2010-10-03',NULL,'pymotw');
INSERT INTO "task" VALUES(2,1,'write about

```

```
random', 'waiting', '2010-10-10', NULL, 'pymotw');
INSERT INTO "task" VALUES(3,1,'write about
sqlite3', 'active', '2010-10-17', NULL, 'pymotw');
DELETE FROM sqlite_sequence;
INSERT INTO "sqlite_sequence" VALUES('task',3);
COMMIT;
```

7.5.13 SQL 中使用 Python 函数

SQL 语法支持在查询中调用函数，可以在列列表中调用，也可以在 select 语句的 where 子句中调用。利用这个特性，从查询返回数据之前可以先处理数据，可以用于在不同格式之间转换、完成一些计算（否则使用纯 SQL 会很麻烦），以及重用应用代码。

```
import sqlite3

db_filename = 'todo.db'

def encrypt(s):
    print 'Encrypting %r' % s
    return s.encode('rot-13')

def decrypt(s):
    print 'Decrypting %r' % s
    return s.encode('rot-13')

with sqlite3.connect(db_filename) as conn:

    conn.create_function('encrypt', 1, encrypt)
    conn.create_function('decrypt', 1, decrypt)
    cursor = conn.cursor()

    # Raw values
    print 'Original values:'
    query = "select id, details from task"
    cursor.execute(query)
    for row in cursor.fetchall():
        print row

    print '\nEncrypting...'
    query = "update task set details = encrypt(details)"
    cursor.execute(query)
    print '\nRaw encrypted values:'
    query = "select id, details from task"
    cursor.execute(query)
    for row in cursor.fetchall():
        print row
```



```

print '\nDecrypting in query...'
query = "select id, decrypt(details) from task"
cursor.execute(query)
for row in cursor.fetchall():
    print row

```

函数使用 Connection 的 create_function() 方法提供。参数包括函数名（即 SQL 中使用的函数名）、函数所取的参数个数，以及要提供的 Python 函数。

```
$ python sqlite3_create_function.py
```

Original values:

```

(1, u'write about select')
(2, u'write about random')
(3, u'write about sqlite3')
(4, u'finish reviewing markup')
(5, u'revise chapter intros')
(6, u'subtitle')

```

Encrypting...

```

Encrypting u'write about select'
Encrypting u'write about random'
Encrypting u'write about sqlite3'
Encrypting u'finish reviewing markup'
Encrypting u'revise chapter intros'
Encrypting u'subtitle'

```

Raw encrypted values:

```

(1, u'jevgr nobhg fryrpg')
(2, u'jevgr nobhg enaqbz')
(3, u'jevgr nobhg fdyvgr3')
(4, u'svavfu erivrjvat znexhc')
(5, u'erivfr puncgre vagebf')
(6, u'fhogvgyr')

```

Decrypting in query...

```

Decrypting u'jevgr nobhg fryrpg'
Decrypting u'jevgr nobhg enaqbz'
Decrypting u'jevgr nobhg fdyvgr3'
Decrypting u'svavfu erivrjvat znexhc'
Decrypting u'erivfr puncgre vagebf'
Decrypting u'fhogvgyr'
(1, u'write about select')
(2, u'write about random')
(3, u'write about sqlite3')
(4, u'finish reviewing markup')
(5, u'revise chapter intros')
(6, u'subtitle')

```



7.5.14 定制聚集

聚集函数会收集多个单独的数据，并以某种方式汇总。avg()(取平均值)、min()、max() 和 count() 都是内置聚集函数的例子。

sqlite3 使用的聚集器 API 定义为一个包含两个方法的类。处理查询时会对各个数据值分别调用一次 step() 方法。finalize() 方法在查询的最后调用一次，并返回聚集值。下面这个例子为 mode 实现了一个聚集器。它会返回输入中出现最频繁的值。

```
import sqlite3
import collections

db_filename = 'todo.db'

class Mode(object):
    def __init__(self):
        self.counter = collections.Counter()
    def step(self, value):
        print 'step(%r)' % value
        self.counter[value] += 1
    def finalize(self):
        result, count = self.counter.most_common(1)[0]
        print 'finalize() -> %r (%d times)' % (result, count)
        return result

with sqlite3.connect(db_filename) as conn:
    conn.create_aggregate('mode', 1, Mode)

    cursor = conn.cursor()
    cursor.execute("""
select mode(deadline) from task where project = 'pymotw'
""")
    row = cursor.fetchone()
    print 'mode(deadline) is:', row[0]
```

聚集器类用 Connection 的 create_aggregate() 方法注册。参数包括函数名（即 SQL 中使用的函数名）、step() 方法所取的参数个数，以及要使用的类。

```
$ python sqlite3_create_aggregate.py

step(u'2010-10-03')
step(u'2010-10-10')
step(u'2010-10-17')
step(u'2010-10-02')
step(u'2010-10-03')
step(u'2010-10-03')
finalize() -> u'2010-10-03' (3 times)
mode(deadline) is: 2010-10-03
```

7.5.15 定制排序

比对 (collation) 是一个比较函数，在 SQL 查询的 order by 部分使用。对于 SQLite 无法在内部排序的数据类型，可以使用定制比对来比较。例如，我们需要一个定制比对来对保存在 sqlite3_custom_type.py 中的 pickle 对象排序。

```
import sqlite3
try:
    import cPickle as pickle
except:
    import pickle

db_filename = 'todo.db'

def adapter_func(obj):
    return pickle.dumps(obj)
def converter_func(data):
    return pickle.loads(data)

class MyObj(object):
    def __init__(self, arg):
        self.arg = arg
    def __str__(self):
        return 'MyObj(%r)' % self.arg
    def __cmp__(self, other):
        return cmp(self.arg, other.arg)

# Register the functions for manipulating the type.
sqlite3.register_adapter(MyObj, adapter_func)
sqlite3.register_converter("MyObj", converter_func)

def collation_func(a, b):
    a_obj = converter_func(a)
    b_obj = converter_func(b)
    print 'collation_func(%s, %s)' % (a_obj, b_obj)
    return cmp(a_obj, b_obj)

with sqlite3.connect(db_filename,
                     detect_types=sqlite3.PARSE_DECLTYPES,
                     ) as conn:
    # Define the collation
    conn.create_collation('unpickle', collation_func)

    # Clear the table and insert new values
    conn.execute('delete from obj')
    conn.executemany('insert into obj (data) values (?)',
                     [(MyObj(x),) for x in xrange(5, 0, -1)],
```

```

    )

    # Query the database for the objects just saved
    print 'Querying:'
    cursor = conn.cursor()
    cursor.execute("""
    select id, data from obj order by data collate unpickle
    """)
    for obj_id, obj in cursor.fetchall():
        print obj_id, obj

```

比对函数的参数是字节串，所以在完成比较之前必须解除 pickle，并转换为 MyObj 实例。

```
$ python sqlite3_create_collation.py
```

```

Querying:
collation_func(MyObj(5), MyObj(4))
collation_func(MyObj(4), MyObj(3))
collation_func(MyObj(4), MyObj(2))
collation_func(MyObj(3), MyObj(2))
collation_func(MyObj(3), MyObj(1))
collation_func(MyObj(2), MyObj(1))
7 MyObj(1)
6 MyObj(2)
5 MyObj(3)
4 MyObj(4)
3 MyObj(5)

```

7.5.16 线程和连接共享

出于历史原因，由于必须使用老版本的 SQLite，Connection 对象不能在线程间共享。每个线程必须创建自己的数据库连接。

```

import sqlite3
import sys
import threading
import time

db_filename = 'todo.db'
isolation_level = None # autocommit mode

def reader(conn):
    my_name = threading.currentThread().name
    print 'Starting thread'
    try:
        cursor = conn.cursor()
        cursor.execute('select * from task')
        results = cursor.fetchall()

```



```

        print 'results fetched'
    except Exception, err:
        print 'ERROR:', err
    return

if __name__ == '__main__':

    with sqlite3.connect(db_filename,
                        isolation_level=isolation_level,
                        ) as conn:
        t = threading.Thread(name='Reader 1',
                            target=reader,
                            args=(conn,))

        t.start()
        t.join()

```

如果试图在线程之间共享一个连接，这会导致一个异常。

```
$ python sqlite3_threading.py
```

```
Starting thread
```

```
ERROR: SQLite objects created in a thread can only be used in that
same thread.The object was created in thread id 4299299872 and
this is thread id 4311166976
```

7.5.17 限制对数据的访问

与其他更大的关系数据库相比，尽管 SQLite 没有用户访问控制，但是确实提供了一种机制来限制列访问。每个连接可以安装一个授权函数（authorizer function），运行时可以根据所需的原则来批准或拒绝访问列。这个授权函数会在解析 SQL 语句时调用，将传入 5 个参数。第一个参数是一个动作码，指示所完成的操作的类型（读、写、删除等等）。其余的参数则取决于动作码。对于 SQLITE_READ 操作，这 4 个参数分别是表名、列名、SQL 语句中访问出现的位置（主查询、触发器等等）和 None。

```

import sqlite3

db_filename = 'todo.db'
def authorizer_func(action, table, column, sql_location, ignore):
    print '\nauthorizer_func(%s, %s, %s, %s, %s)' % \
        (action, table, column, sql_location, ignore)

response = sqlite3.SQLITE_OK # be permissive by default

if action == sqlite3.SQLITE_SELECT:
    print 'requesting permission to run a select statement'
    response = sqlite3.SQLITE_OK

```

```

elif action == sqlite3.SQLITE_READ:
    print 'requesting access to column %s.%s from %s' % \
        (table, column, sql_location)
    if column == 'details':
        print ' ignoring details column'
        response = sqlite3.SQLITE_IGNORE
    elif column == 'priority':
        print ' preventing access to priority column'
        response = sqlite3.SQLITE_DENY

return response

with sqlite3.connect(db_filename) as conn:
    conn.row_factory = sqlite3.Row
    conn.set_authorizer(authorizer_func)

    print 'Using SQLITE_IGNORE to mask a column value:'
    cursor = conn.cursor()
    cursor.execute("""
select id, details from task where project = 'pymotw'
""")
    for row in cursor.fetchall():
        print row['id'], row['details']

    print '\nUsing SQLITE_DENY to deny access to a column:'
    cursor.execute("""
select id, priority from task where project = 'pymotw'
""")
    for row in cursor.fetchall():
        print row['id'], row['details']

```

这个例子使用了 `SQLITE_IGNORE`，从而在查询结果中将从 `task.details` 列得到的串替换为 `null` 值。通过返回 `SQLITE_DENY`，还将避免对 `task.priority` 列的访问，如果尝试访问 `task.priority` 列，则会导致 SQLite 产生一个异常。

```
$ python sqlite3_set_authorizer.py
```

```
Using SQLITE_IGNORE to mask a column value:
```

```
authorizer_func(21, None, None, None, None)
requesting permission to run a select statement
```

```
authorizer_func(20, task, id, main, None)
requesting access to column task.id from main
```

```
authorizer_func(20, task, details, main, None)
requesting access to column task.details from main
```

```

    ignoring details column

authorizer_func(20, task, project, main, None)
requesting access to column task.project from main
1 None
2 None
3 None
4 None
5 None
6 None

Using SQLITE_DENY to deny access to a column:

authorizer_func(21, None, None, None, None)
requesting permission to run a select statement

authorizer_func(20, task, id, main, None)
requesting access to column task.id from main

authorizer_func(20, task, priority, main, None)
requesting access to column task.priority from main
preventing access to priority column
Traceback (most recent call last):
  File "sqlite3_set_authorizer.py", line 51, in <module>
    """
sqlite3.DatabaseError: access to task.priority is prohibited

```

sqlite3 中提供了一些可用的动作码，它们都作为常量提供，名字前都有前缀 `SQLITE_`。每一类 SQL 语句可以加标志，也可以控制对单个列的访问。

参见：

sqlite3 (<http://docs.python.org/library/sqlite3.html>) 这个模块的标准库文档。

PEP 249 (www.python.org/dev/peps/pep-0249)—DB API 2.0 Specification 这是访问关系数据库的模块的一个标准接口。

SQLite (www.sqlite.org/) SQLite 库的官方网站。

shelve (7.2 节) 用于保存任意 Python 对象的键-值库。

SQLAlchemy (<http://sqlalchemy.org/>) 一个流行的对象关系映射器，除了很多其他关系数据库外，还支持 SQLite。

7.6 xml.etree.ElementTree——XML 操纵 API

作用：生成和解析 XML 文档。

Python 版本：2.5 及以后版本

ElementTree 库包括一些工具，可以使用基于事件和基于文档的 API 解析 XML、用 XPath

表达式搜索已解析的文档，还可以创建或修改现有文档。

注意：为简单起见，本节中的所有例子都使用 ElementTree 的 Python 实现，不过 xml.etree.cElementTree 中还有一个 C 实现。

7.6.1 解析 XML 文档

已解析的 XML 文档在内存中由 ElementTree 和 Element 对象表示，这些对象基于 XML 文档中节点嵌套的方式以树结构相互连接。

用 parse() 解析一个完整的文档时，会返回一个 ElementTree 实例。这个树了解输入文档中的所有数据，另外可以原地搜索或操纵树中的节点。基于这种灵活性，可以更为方便地处理已解析文档，不过，与基于事件的解析方法相比，这样往往需要更多的内存，因为整个文档必须一次全部加载。

对于简单的小文档（如以下播客列表，表示为一个 OPML 大纲），其内存需求并不大：

```
<?xml version="1.0" encoding="UTF-8"?>
<opml version="1.0">
  <head>
    <title>My Podcasts</title>
    <dateCreated>Sun, 07 Mar 2010 15:53:26 GMT</dateCreated>
    <dateModified>Sun, 07 Mar 2010 15:53:26 GMT</dateModified>
  </head>
  <body>
    <outline text="Fiction">
      <outline
        text="tor.com / category / tordotstories" type="rss"
        xmlUrl="http://www.tor.com/rss/category/TorDotStories"
        htmlUrl="http://www.tor.com/" />
    </outline>
    <outline text="Python">
      <outline
        text="PyCon Podcast" type="rss"
        xmlUrl="http://advocacy.python.org/podcasts/pycon.rss"
        htmlUrl="http://advocacy.python.org/podcasts/" />
      <outline
        text="A Little Bit of Python" type="rss"
        xmlUrl="http://advocacy.python.org/podcasts/littlebit.rss"
        htmlUrl="http://advocacy.python.org/podcasts/" />
    </outline>
  </body>
</opml>
```

要解析这个文档，需要向 parse() 传递一个打开的文件句柄。

```
from xml.etree import ElementTree
```



```
with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)
```

```
print tree
```

它会读取数据、解析 XML，并返回一个 ElementTree 对象。

```
$ python ElementTree_parse_opml.py
```

```
<xml.etree.ElementTree.ElementTree object at 0x100dca350>
```

7.6.2 遍历解析树

要按顺序访问所有子节点，可以使用 iter() 创建一个生成器，迭代处理这个 ElementTree 实例。

```
from xml.etree import ElementTree
import pprint
```

```
with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)
```

```
for node in tree.iter():
    print node.tag
```

这个例子会打印整个树，一次打印一个标记。

```
$ python ElementTree_dump_opml.py
```

```
opml
head
title
dateCreated
dateModified
body
outline
outline
outline
outline
outline
```

如果只是打印播客的名字组和提要 URL，可以只迭代处理 outline 节点（而不考虑首部中的所有数据），通过查找 attrib 字典中的值来打印 text 和 xmlUrl 属性。

```
from xml.etree import ElementTree
```

```
with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)
```

```
for node in tree.iter('outline'):
    name = node.attrib.get('text')
```

```

url = node.attrib.get('xmlUrl')
if name and url:
    print ' %s' % name
    print ' %s' % url
else:
    print name

```

iter() 的 'outline' 参数意味着只处理标记为 'outline' 的节点。

```
$ python ElementTree_show_feed_urls.py
```

```

Fiction
tor.com / category / tordotstories
http://www.tor.com/rss/category/TorDotStories
Python
PyCon Podcast
http://advocacy.python.org/podcasts/pycon.rss
A Little Bit of Python
http://advocacy.python.org/podcasts/littlebit.rss

```

7.6.3 查找文档中的节点

如果像这样查看整个树并搜索有关的节点，可能很容易出错。前面的例子必须查看每一个 outline 节点，来确定这是一个组（只有一个 text 属性的节点）还是一个播客（包含 text 和 xmlUrl 的节点）。要生成一个简单的播客提要 URL 列表，而不包括名字或组，可以简化逻辑，使用 findall() 来查找包含更多描述性搜索特性的节点。

对于以上的第一个版本，作为第一次转换，可以用一个 XPath 参数来查找所有 outline 节点。

```

from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.findall('.//outline'):
    url = node.attrib.get('xmlUrl')
    if url:
        print url

```

这个版本中的逻辑与使用 getiterator() 的版本并没有显著区别。这里仍然必须检查 URL 是否存在，只不过如果没有发现 URL，它不会打印组名。

```

$ python ElementTree_find_feeds_by_tag.py
http://www.tor.com/rss/category/TorDotStories
http://advocacy.python.org/podcasts/pycon.rss
http://advocacy.python.org/podcasts/littlebit.rss

```

outline 节点只有 2 层嵌套，可以利用这一点。把搜索路径修改为 //outline/outline，这意味着只处理 outline 节点的第 2 层。

```

from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.findall('.//outline/outline'):
    url = node.attrib.get('xmlUrl')
    print url

```

输入中所有嵌套深度为2层的 outline 节点都认为有一个 xmlURL 属性指向播客提要，所以循环在使用这个属性之前可以不再先做检查。

```

$ python ElementTree_find_feeds_by_structure.py

http://www.tor.com/rss/category/TorDotStories
http://advocacy.python.org/podcasts/pycon.rss
http://advocacy.python.org/podcasts/littlebit.rss

```

不过，这个版本仅限于当前的这个结构，所以如果 outline 节点重新组织为一个更深的树，这个版本就无法正常工作。

7.6.4 解析节点属性

findall() 和 iter() 返回的元素是 Element 对象，各个对象分别表示 XML 解析树中的一个节点。每个 Element 有一些属性来访问 XML 中的数据。可以用一个稍有些牵强的示例输入文件 data.xml 来说明。

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <top>
3   <child>Regular text.</child>
4   <child_with_tail>Regular text.</child_with_tail>"Tail" text.
5   <with_attributes name="value" foo="bar" />
6   <entity_expansion attribute="This &#38; That">
7     That &#38; This
8   </entity_expansion>
9 </top>

```

节点的属性可以由 attrib 属性得到，它就像是一个字典。

```

from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

node = tree.find('.//with_attributes')
print node.tag
for name, value in sorted(node.attrib.items()):
    print ' %4s = "%s"' % (name, value)

```

输入文件第 5 行上的节点有两个属性 `name` 和 `foo`。

```
$ python ElementTree_node_attributes.py
```

```
with_attributes
    foo = "bar"
    name = "value"
```

还可以得到节点的文本内容，以及结束标记后面的 `tail` 文本[⊖]。

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

for path in [ './child', './child_with_tail' ]:
    node = tree.find(path)
    print node.tag
    print '  child node text:', node.text
    print '  and tail text  :', node.tail
```

第 3 行的 `child` 节点包含嵌入文本，第 4 行的节点除了文本还有 `tail` 文本（包括空白符）。

```
$ python ElementTree_node_text.py
```

```
child
  child node text: Regular text.
  and tail text  :

child_with_tail
  child node text: Regular text.
  and tail text  : "Tail" text.
```

返回值之前，文档中嵌入的 XML 实体引用会转换为适当的字符。

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

node = tree.find('entity_expansion')
print node.tag
print '  in attribute:', node.attrib['attribute']
print '  in text      :', node.text.strip()
```

这个自动转换意味着可以忽略表示 XML 文档中某些字符的实现细节。

```
$ python ElementTree_entity_references.py
```

⊖ `tail` 文本是指位于结束标记之后，下一元素开始或父元素结束之前的所有文本。——译者注

```
entity_expansion
  in attribute: This & That
  in text      : That & This
```

7.6.5 解析时监视事件

另一个处理 XML 文档的 API 是基于事件的。解析器为开始标记生成 start 事件，为结束标记生成 end 事件。解析阶段中可以通过迭代处理事件流从文档抽取数据，如果以后没有必要处理整个文档，或者没有必要将整个解析文档都保存在内存中，基于事件的 API 就会很方便。

有以下事件类型：

start 遇到一个新标记。会处理标记的结束尖括号，但不处理内容。

end 已经处理结束标记的结束尖括号。所有子节点都已经处理。

start-ns 开始一个命名空间声明。

end-ns 结束一个命名空间声明。

iterparse() 返回一个可迭代的对象，它将生成元组，其中包含事件名和触发事件的节点。

```
depth = 0
prefix_width = 8
prefix_dots = '.' * prefix_width
line_template = ''.join(['{prefix:<0.{prefix_len}}',
                          '{event:<8}',
                          '{suffix:<{suffix_len}} ',
                          '{node.tag:<l2} ',
                          '{node_id}',
                          ])

EVENT_NAMES = ['start', 'end', 'start-ns', 'end-ns']

for (event, node) in iterparse('podcasts.opml', EVENT_NAMES):
    if event == 'end':
        depth -= 1

    prefix_len = depth * 2

    print line_template.format(
        prefix=prefix_dots,
        prefix_len=prefix_len,
        suffix='',
        suffix_len=(prefix_width - prefix_len),
        node=node,
        node_id=id(node),
        event=event,
    )

    if event == 'start':
        depth += 1
```



默认情况下，只会生成 end 事件。要查看其他事件，可以将所需的事件名列表传入 `iterparse()`，如下例所示。

```
$ python ElementTree_show_all_events.py

start          opml          4309429072
..start        head          4309429136
....start      title        4309429200
....end        title        4309429200
....start      dateCreated  4309429392
....end        dateCreated  4309429392
....start      dateModified 4309429584
....end        dateModified 4309429584
..end          head          4309429136
..start        body          4309429968
....start      outline      4309430032
.....start    outline      4309430096
.....end      outline      4309430096
....end        outline      4309430032
....start      outline      4309430160
.....start    outline      4309430224
.....end      outline      4309430224
.....start    outline      4309459024
.....end      outline      4309459024
....end        outline      4309430160
..end          body          4309429968
end            opml          4309429072
```

以事件方式进行处理对于某些操作来说更为自然，如将 XML 输入转换为另外某种格式。可以使用这个技术将播客列表（来自前面的例子）由 XML 文件转换为一个 CSV 文件，从而可以将它们加载到一个电子表格或数据库应用。

```
import csv
from xml.etree.ElementTree import iterparse
import sys

writer = csv.writer(sys.stdout, quoting=csv.QUOTE_NONNUMERIC)

group_name = ''

for (event, node) in iterparse('podcasts.opml', events=['start']):
    if node.tag != 'outline':
        # Ignore anything not part of the outline
        continue
    if not node.attrib.get('xmlUrl'):
        # Remember the current group
        group_name = node.attrib['text']
```

```

else:
    # Output a podcast entry
    writer.writerow( (group_name, node.attrib['text'],
                     node.attrib['xmlUrl'],
                     node.attrib.get('htmlUrl', ''),
                     )
                    )

```

这个转换程序并不需要将整个已解析的输入文件保存在内存中，而且遇到输入中的各个节点时才进行处理，这样做也更为高效。

```
$ python ElementTree_write_podcast_csv.py
```

```

"Fiction", "tor.com / category / tordotstories", "http://www.tor.com/r\
ss/category/TorDotStories", "http://www.tor.com/"
"Python", "PyCon Podcast", "http://advocacy.python.org/podcasts/pycon.\
rss", "http://advocacy.python.org/podcasts/"
"Python", "A Little Bit of Python", "http://advocacy.python.org/podcas\
ts/littlebit.rss", "http://advocacy.python.org/podcasts/"

```

注意: 这里重新调整了 `ElementTree_write_podcast_csv.py` 的输出格式，以适合本页显示。以 \ 结尾的输出行指示这是一个人为换行。

7.6.6 创建一个定制树构造器

要处理解析事件，一种可能更为高效的方法是将标准的树构造器行为替换为一种定制行为。`ElementTree` 解析器使用一个 `XMLTreeBuilder` 处理 XML，并调用目标类的方法来保存结果。通常输出是由默认 `TreeBuilder` 类创建的一个 `ElementTree` 实例。可以将 `TreeBuilder` 替换为另一个类，使它在实例化 `Element` 节点之前接收事件，从而节省这部分开销。

可以将上一节的 XML-CSV 转换器重新实现为一个树构造器。

```

import csv
from xml.etree.ElementTree import XMLTreeBuilder
import sys

class PodcastListToCSV(object):

    def __init__(self, outputFile):
        self.writer = csv.writer(outputFile,
                                quoting=csv.QUOTE_NONNUMERIC)

        self.group_name = ''
        return

    def start(self, tag, attrib):
        if tag != 'outline':
            # Ignore anything not part of the outline

```

```

        return
    if not attrib.get('xmlUrl'):
        # Remember the current group
        self.group_name = attrib['text']
    else:
        # Output a podcast entry
        self.writer.writerow( (self.group_name, attrib['text'],
                               attrib['xmlUrl'],
                               attrib.get('htmlUrl', ''),
                               )
                               )

def end(self, tag):
    # Ignore closing tags
    pass
def data(self, data):
    # Ignore data inside nodes
    pass
def close(self):
    # Nothing special to do here
    return

target = PodcastListToCSV(sys.stdout)
parser = XMLTreeBuilder(target=target)
with open('podcasts.opml', 'rt') as f:
    for line in f:
        parser.feed(line)
parser.close()

```

PodcastListToCSV 实现了 TreeBuilder 协议。每次遇到一个新的 XML 标记时，会调用 start() 并提供标记名和属性。看到一个结束标记时，则会根据标记名调用 end()。在这二者之间，如果一个节点有内容会调用 data()（一般认为树构造器会跟踪“当前”节点）。所有输入都已处理时，将调用 close()。它可能返回一个值，这会返回给 XMLTreeBuilder 用户。

```
$ python ElementTree_podcast_csv_treebuilder.py
```

```

"Fiction", "tor.com / category / tordotstories", "http://www.tor.com/rss/category/TorDotStories", "http://www.tor.com/"
"Python", "PyCon Podcast", "http://advocacy.python.org/podcasts/pycon.rss", "http://advocacy.python.org/podcasts/"
"Python", "A Little Bit of Python", "http://advocacy.python.org/podcasts/littlebit.rss", "http://advocacy.python.org/podcasts/"

```

注意：这里重新调整了 ElementTree_podcast_csv_treebuilder.py 的输出格式，以适合本页显示。以 \ 结尾的输出行指示这是一个人为换行。

7.6.7 解析串

如果只是处理少量的 XML 文本，特别是可能嵌入在程序源代码中的字符串字面量，可以使用 XML()，将包含待解析 XML 的字符串作为其惟一参数。

```
from xml.etree.ElementTree import XML

parsed = XML('''
<root>
  <group>
    <child id="a">This is child "a".</child>
    <child id="b">This is child "b".</child>
  </group>
  <group>
    <child id="c">This is child "c".</child>
  </group>
</root>
''')

print 'parsed =', parsed
def show_node(node):
    print node.tag
    if node.text is not None and node.text.strip():
        print '  text: "%s"' % node.text
    if node.tail is not None and node.tail.strip():
        print '  tail: "%s"' % node.tail
    for name, value in sorted(node.attrib.items()):
        print '  %-4s = "%s"' % (name, value)
    for child in node:
        show_node(child)
    return

for elem in parsed:
    show_node(elem)
```

与 parse() 不同，这个函数的返回值是一个 Element 实例而不是 ElementTree。Element 直接支持迭代器协议，所以没有必要调用 getiterator()。

```
$ python ElementTree_XML.py

parsed = <Element 'root' at 0x100dcba50>
group
child
  text: "This is child "a"."
  id  = "a"
child
  text: "This is child "b"."
  id  = "b"
```

```

group
child
    text: "This is child "c"."
    id   = "c"

```

对于使用 id 属性来标识相应惟一节点的结构化 XML，可以使用便利方法 XMLID() 来访问解析结果。

```

from xml.etree.ElementTree import XMLID

tree, id_map = XMLID('''
<root>
  <group>
    <child id="a">This is child "a".</child>
    <child id="b">This is child "b".</child>
  </group>
  <group>
    <child id="c">This is child "c".</child>
  </group>
</root>
''')

for key, value in sorted(id_map.items()):
    print '%s = %s' % (key, value)

```

XMLID() 将解析树作为一个 Element 对象返回，并提供一个字典，将 id 属性串映射到树中的单个节点。

```

$ python ElementTree_XMLID.py

a = <Element 'child' at 0x100dcab90>
b = <Element 'child' at 0x100dcac50>
c = <Element 'child' at 0x100dcae90>

```

参见：

Outline Processor Markup Language, OPML (<http://www.opml.org/>) Dave Winer 的 OPML 规范和文档。

XML Path Language, XPath (<http://www.w3.org/TR/xpath/>) 标识 XML 文档中各部分的一种语法。

XPath Support in ElementTree (<http://effbot.org/zone/element-xpath.htm>) Fredrick Lundh 提供的 ElementTree 原始文档的一部分。

csv (7.7 节) 读写逗号分隔值文件。

7.6.8 用元素节点构造文档

除解析功能外，xml.etree.ElementTree 还支持由应用中构造的 Element 对象创建良构的 XML 文档。解析文档时使用的 Element 类还知道如何生成其内容的一个串行化形式，然后可以

将这个串行化内容写至一个文件或另一个数据流。

有3个辅助函数对于创建 Element 节点层次结构很有用。Element() 创建一个标准节点，SubElement() 将一个新节点关联到一个父节点，Comment() 创建一个节点，它会使用 XML 的注释语法串行化。

```
from xml.etree.ElementTree import ( Element,
                                    SubElement,
                                    Comment,
                                    tostring,
                                    )

top = Element('top')

comment = Comment('Generated for PyMOTW')
top.append(comment)

child = SubElement(top, 'child')
child.text = 'This child contains text.'

child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has regular text.'
child_with_tail.tail = 'And "tail" text.'

child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'

print tostring(top)
```

这个输出只包含树中的 XML 节点，而不包括含版本和编码的 XML 声明。

```
$ python ElementTree_create.py
```

```
<top><!--Generated for PyMOTW--><child>This child contains text.</child>
<child_with_tail>This child has regular text.</child_with_tail>And
"tail" text.<child_with_entity_ref>This & that</child_with_entity_ref></top>
```

child_with_entity_ref 文本中的 & 字符会自动转换为实体引用 &#x26;

7.6.9 美观打印 XML

ElementTree 不会格式化 tostring() 的输出，所以很易读，因为增加额外的空白符会改变文档的内容。为了让输出更易懂，后面的例子将使用 xml.dom.minidom 重新解析 XML，然后使用其 toprettyxml() 方法。

```
from xml.etree import ElementTree
from xml.dom import minidom
```

```
def prettify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent="  ")
```

现在更新后的例子如下所示:

```
from xml.etree.ElementTree import Element, SubElement, Comment
from ElementTree pretty import prettify
```

```
top = Element('top')
```

```
comment = Comment('Generated for PyMOTW')
top.append(comment)
```

```
child = SubElement(top, 'child')
child.text = 'This child contains text.'
```

```
child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has regular text.'
child_with_tail.tail = 'And "tail" text.'
```

```
child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'
```

```
print prettify(top)
```

输出将更易读。

```
$ python ElementTree_create_pretty.py
```

```
<?xml version="1.0" ?>
<top>
  <!--Generated for PyMOTW-->
  <child>
    This child contains text.
  </child>
  <child_with_tail>
    This child has regular text.
  </child_with_tail>
  And &quot;tail&quot; text.
  <child_with_entity_ref>
    This &amp; that
  </child_with_entity_ref>
</top>
```



除了添加用于格式化的额外空白符，xml.dom.minidom 美观打印器还会向输出添加一个 XML 声明。

7.6.10 设置元素属性

前面的例子都由标记和文本内容来创建节点，但是没有设置节点的任何属性。7.6.1 节中的很多例子都在处理一个列举播客及其提要的 OPML 文件。树中的 outline 节点使用了对应组名和播客特性的属性。可以用 ElementTree 由一个 CSV 输入文件构造一个类似的 XML 文件，并在构造树时设置所有元素属性。

```
import csv
from xml.etree.ElementTree import ( Element,
                                    SubElement,
                                    Comment,
                                    tostring,
                                    )

import datetime
from ElementTree_pretty import prettify

generated_on = str(datetime.datetime.now())

# Configure one attribute with set()
root = Element('opml')
root.set('version', '1.0')

root.append(
    Comment('Generated by ElementTree_csv_to_xml.py for PyMOTW')
)

head = SubElement(root, 'head')
title = SubElement(head, 'title')
title.text = 'My Podcasts'
dc = SubElement(head, 'dateCreated')
dc.text = generated_on
dm = SubElement(head, 'dateModified')
dm.text = generated_on

body = SubElement(root, 'body')

with open('podcasts.csv', 'rt') as f:
    current_group = None
    reader = csv.reader(f)
    for row in reader:
        group_name, podcast_name, xml_url, html_url = row
        if current_group is None or group_name != current_group.text:
            # Start a new group
```

```

        current_group = SubElement(body, 'outline',
                                    {'text':group_name})
    # Add this podcast to the group,
    # setting all its attributes at
    # once.
    podcast = SubElement(current_group, 'outline',
                          {'text':podcast_name,
                           'xmlUrl':xml_url,
                           'htmlUrl':html_url,
                          })

```

```
print prettify(root)
```

这个例子使用两种技术来设置新节点的属性值。根节点用 `set()` 配置，一次修改一个属性。另外通过向节点工厂传入一个字典，对播客节点一次给定所有属性。

```
$ python ElementTree_csv_to_xml.py
```

```

<?xml version="1.0" ?>
<opml version="1.0">
  <!--Generated by ElementTree_csv_to_xml.py for PyMOTW-->
  <head>
    <title>
      My Podcasts
    </title>
    <dateCreated>
      2010-12-03 08:48:58.065172
    </dateCreated>
    <dateModified>
      2010-12-03 08:48:58.065172
    </dateModified>
  </head>
  <body>
    <outline text="Books and Fiction">
      <outline htmlUrl="http://www.tor.com/" text="tor.com / categor
y / tordotstories" xmlUrl="http://www.tor.com/rss/category/TorDotSto
ries"/>
    </outline>
    <outline text="Python">
      <outline htmlUrl="http://advocacy.python.org/podcasts/" text="
PyCon Podcast" xmlUrl="http://advocacy.python.org/podcasts/pycon.rss
"/>
    </outline>
    <outline text="Python">
      <outline htmlUrl="http://advocacy.python.org/podcasts/" text="
A Little Bit of Python" xmlUrl="http://advocacy.python.org/podcasts/
littlebit.rss"/>

```

```

    </outline>
    <outline text="Python">
        <outline htmlUrl="" text="Django Dose Everything Feed" xmlUrl=
"http://djangodose.com/everything/feed/" />
    </outline>
</body>
</opml>

```

7.6.11 由节点列表构造树

利用 `extend()` 方法可以将多个子节点一同添加到一个 `Element` 实例。`extend()` 的参数可以是任意可迭代对象，包括 `list` 或另一个 `Element` 实例。

```

from xml.etree.ElementTree import Element, tostring
from ElementTree_pretty import prettify

top = Element('top')

children = [
    Element('child', num=str(i))
    for i in xrange(3)
]

top.extend(children)

print prettify(top)

```

给定一个 `list` 时，列表中的节点会直接添加到新的父节点中。

```
$ python ElementTree_extend.py
```

```

<?xml version="1.0" ?>
<top>
  <child num="0"/>
  <child num="1"/>
  <child num="2"/>
</top>

```

给定另一个 `Element` 实例时，该节点的子节点会添加到新的父节点中。

```

from xml.etree.ElementTree import Element, SubElement, tostring, XML
from ElementTree_pretty import prettify

top = Element('top')

parent = SubElement(top, 'parent')

children = XML(
    '<root><child num="0" /><child num="1" /><child num="2" /></root>'
)

```

```

    )
    parent.extend(children)

    print prettify(top)

```

在这个例子中，通过解析 XML 串创建的 root 节点有 3 个子节点，它们都添加到 parent 节点中。root 节点不是输出树的一部分。

```
$ python ElementTree_extend_node.py
```

```

<?xml version="1.0" ?>
<top>
  <parent>
    <child num="0"/>
    <child num="1"/>
    <child num="2"/>
  </parent>
</top>

```

要了解的重要一点是，`extend()` 并不改变节点现有的父子关系。如果传入 `extend()` 的值已经存在于树中的某个位置，它们将仍在原处，并在输出中重复。

```

from xml.etree.ElementTree import Element, SubElement, tostring, XML
from ElementTree_pretty import prettify

top = Element('top')

parent_a = SubElement(top, 'parent', id='A')
parent_b = SubElement(top, 'parent', id='B')

# Create children
children = XML(
    '<root><child num="0" /><child num="1" /><child num="2" /></root>'
)

# Set the id to the Python object id of the node
# to make duplicates easier to spot.
for c in children:
    c.set('id', str(id(c)))

# Add to first parent
parent_a.extend(children)

print 'A:'
print prettify(top)
print

# Copy nodes to second parent

```




```
parent_b.extend(children)
```

```
print 'B:'
print prettify(top)
print
```

这里将子节点的 id 属性设置为 Python 惟一对象标识符，由此强调同样的节点对象可以在输出树中出现多次。

```
$ python ElementTree_extend_node_copy.py
```

```
A:
<?xml version="1.0" ?>
<top>
  <parent id="A">
    <child id="4309786256" num="0"/>
    <child id="4309786320" num="1"/>
    <child id="4309786512" num="2"/>
  </parent>
  <parent id="B"/>
</top>
```

```
B:
<?xml version="1.0" ?>
<top>
  <parent id="A">
    <child id="4309786256" num="0"/>
    <child id="4309786320" num="1"/>
    <child id="4309786512" num="2"/>
  </parent>
  <parent id="B">
    <child id="4309786256" num="0"/>
    <child id="4309786320" num="1"/>
    <child id="4309786512" num="2"/>
  </parent>
</top>
```

7.6.12 将 XML 串行化至一个流

tostring() 实现为将内容写至一个类文件的内存中对象，然后返回一个表示整个元素树的串。处理大量数据时，这种做法需要的内存较少，而且可以更高效地利用 I/O 库，使用 ElementTree 的 write() 方法直接写至一个文件句柄。

```
import sys
from xml.etree.ElementTree import ( Element,
                                     SubElement,
```

```

        Comment,
        ElementTree,
    )

    top = Element('top')

    comment = Comment('Generated for PyMOTW')
    top.append(comment)

    child = SubElement(top, 'child')
    child.text = 'This child contains text.'

    child_with_tail = SubElement(top, 'child_with_tail')
    child_with_tail.text = 'This child has regular text.'
    child_with_tail.tail = 'And "tail" text.'

    child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
    child_with_entity_ref.text = 'This & that'

    empty_child = SubElement(top, 'empty_child')

    ElementTree(top).write(sys.stdout)

```

这个例子使用 `sys.stdout` 写至控制台，不过也可以写至一个打开的文件或套接字。

```
$ python ElementTree_write.py
```

```

<top><!--Generated for PyMOTW--><child>This child contains text.</ch
ild><child_with_tail>This child has regular text.</child_with_tail>A
nd "tail" text.<child_with_entity_ref>This & that</child_with_en
tity_ref><empty_child /></top>

```

树中最后一个节点不包含文本或子节点，所以它写为一个空标记 `<empty_child />`。 `write()` 有一个方法（method）参数，用来控制空节点的处理。

```

import sys
from xml.etree.ElementTree import Element, SubElement, ElementTree

top = Element('top')
child = SubElement(top, 'child')
child.text = 'Contains text.'

empty_child = SubElement(top, 'empty_child')

for method in ['xml', 'html', 'text']:
    print method
    ElementTree(top).write(sys.stdout, method=method)
    print '\n'

```

这里支持 3 个方法。

xml 默认方法，生成 `<empty_child />`。

html 生成标记对，HTML 文档要求必须采用这种方法 (`<empty_child></empty_child>`)。

text 只打印节点的文本，完全跳过空标记。

```
$ python ElementTree_write_method.py
```

```
xml
<top><child>Contains text.</child><empty_child /></top>

html
<top><child>Contains text.</child><empty_child></empty_child></top>

text
Contains text.
```

参见：

Outline Processor Markup Language, OPML (www.opml.org/) Dave Winer 的 OPML 规范和文档。

Pretty-Print XML with Python—Indenting XML(<http://renesd.blogspot.com/2007/05/pretty-print-xml-with-python.html>) Rene Dudfield 关于在 Python 中美观打印 XML 的一些提示。

xml.etree.ElementTree (<http://docs.python.org/library/xml.etree.elementtree.html>) 这个模块的标准库文档。

ElementTree Overview (<http://effbot.org/zone/element-index.htm>) Fredrick Lundh 的原始文档，以及 ElementTree 库开发版的链接。

Process XML in Python with ElementTree(<http://www.ibm.com/developerworks/library/x-matters28/>) David Mertz 的 IBM Developer-Works 文章。

lxml.etree (<http://codespeak.net/lxml/>) 基于 libxml2 的另一个 ElementTree API 实现，提供了更完备的 XPath 支持。

7.7 csv——逗号分隔值文件

作用：读写逗号分隔值文件。

Python 版本：2.3 及以后版本

可以用 csv 模块处理从电子表格和数据库导出的数据，并写入采用字段和记录格式的文本文件，这种格式通常称为逗号分隔值 (comma-separated value, CSV) 格式，因为常用逗号来分隔记录中的字段。

注意：Python 2.5 版本中的 csv 不支持 Unicode 数据。另外处理 ASCII NUL 字符也有问题。建议使用 UTF-8 或可打印的 ASCII 字符。

7.7.1 读文件

可以使用 `reader()` 创建一个对象从 CSV 文件读取数据。这个阅读器可以用作一个迭代器，按顺序处理文件中的行。例如：

```
import csv
import sys

with open(sys.argv[1], 'rt') as f:
    reader = csv.reader(f)
    for row in reader:
        print row
```

`reader()` 的第一个参数是文本行的源。在这个例子中，这是一个文件，不过也可以是任何可迭代的对象（如 `StringIO` 实例、`list` 等等）。还可以指定其他可选参数，来控制如何解析输入数据。

```
"Title 1","Title 2","Title 3"
1,"a",08/18/07
2,"b",08/19/07
3,"c",08/20/07
```

读文件时，输入数据的每一行都会解析，并转换为一个字符串 `list`。

```
$ python csv_reader.py testdata.csv
```

```
['Title 1', 'Title 2', 'Title 3']
['1', 'a', '08/18/07']
['2', 'b', '08/19/07']
['3', 'c', '08/20/07']
```

这个解析器会处理嵌在行字符串中的换行符，正是因为这个原因，这里的“行”（`row`）并不一定等同于文件的一个输入“行”（`line`）。

```
"Title 1","Title 2","Title 3"
1,"first line
second line",08/18/07
```

由解析器返回时，输入中带换行符的字段仍保留内部换行符。

```
$ python csv_reader.py testlinebreak.csv
```

```
['Title 1', 'Title 2', 'Title 3']
['1', 'first line\nsecond line', '08/18/07']
```

7.7.2 写文件

写 CSV 文件与读 CSV 文件同样容易。可以使用 `writer()` 创建一个对象来写数据，然后使用 `writerow()` 迭代处理文本行进行打印。

```

import csv
import sys

with open(sys.argv[1], 'wt') as f:
    writer = csv.writer(f)
    writer.writerow( ('Title 1', 'Title 2', 'Title 3') )
    for i in range(3):
        writer.writerow( (i+1,
                           chr(ord('a') + i),
                           '08/%02d/07' % (i+1),
                           )
                           )

print open(sys.argv[1], 'rt').read()

```

这里的输出与阅读器示例中使用的导出数据看上去不完全相同。

```
$ python csv_writer.py testout.csv
```

```

Title 1,Title 2,Title 3
1,a,08/01/07
2,b,08/02/07
3,c,08/03/07

```

引号

对于书写器，默认的引号行为有所不同，所以前例中第2列和第3列没有加引号。要加引号，需要将 `quoting` 参数设置为另外某种引号模式。

```
writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
```

在这里，`QUOTE_NONNUMERIC` 会在所有包含非数值内容的列周围加引号。

```
$ python csv_writer_quoted.py testout_quoted.csv
```

```

"Title 1","Title 2","Title 3"
1,"a","08/01/07"
2,"b","08/02/07"
3,"c","08/03/07"

```

有4种不同的引号选项，在 `csv` 模块中定义为4个常量。

`QUOTE_ALL` 不论类型是什么，对所有字段都加引号。

`QUOTE_MINIMAL` 对包含特殊字符的字段加引号（所谓特殊字符是指，对于一个用相同方言和选项配置的解析器，可能会造成混淆的字符）。这是默认选项。

`QUOTE_NONNUMERIC` 对所有非整数或浮点数的字段加引号。在阅读器中使用时，不加引号的输入字段会转换为浮点数。

`QUOTE_NONE` 输出中所有内容都不加引号。在阅读器中使用时，引号字符包含在字段值中（正常情况下，它们会处理为定界符并去除）。

7.7.3 方言

逗号分隔值文件没有明确定义的标准，所以解析器必须很灵活。这种灵活性意味着可以用很多参数来控制 csv 如何解析或写数据。并不是将各个参数单独传入阅读器和书写器，可以把它们成组在一起构成一个方言 (dialect) 对象。

Dialect 类可以按名注册，这样 csv 模块的调用者就不需要提前知道参数设置。可以用 `list_dialects()` 获取完整的已注册方言列表。

```
import csv

print csv.list_dialects()
```

这个标准库包括两个方言：excel 和 excel-tabs。excel 方言用于处理采用 Microsoft Excel 默认导出格式的数据，也可以处理 OpenOffice 或 NeoOffice。

```
$ python csv_list_dialects.py
```

```
['excel-tab', 'excel']
```

创建方言

可以不使用逗号来分隔字段，输入文件使用了竖线 (`|`)，如下所示：

```
"Title 1"|"Title 2"|"Title 3"
1|"first line
second line"|08/18/07
```

可以使用适当的定界符注册一个新的方言。

```
import csv

csv.register_dialect('pipes', delimiter='|')

with open('testdata.pipes', 'r') as f:
    reader = csv.reader(f, dialect='pipes')
    for row in reader:
        print row
```

使用 “pipes” 方言，可以像逗号定界文件一样读取文件。

```
$ python csv_dialect.py
```

```
['Title 1', 'Title 2', 'Title 3']
['1', 'first line\nsecond line', '08/18/07']
```

方言参数

方言指定了解析或写一个数据文件时使用的所有记号 (token)。表 7.3 列出了可以指定的文件格式的各个方面，从如何对列定界到使用哪个字符来转义一个记号都可以指定。

表 7.3 CSV 方言参数

属 性	默 认 值	含 义
delimiter	,	字段分隔符（一个字符）
doublequote	True	这个标志控制 quotechar 实例是否成对
escapechar	None	这个字符用来指示一个转义序列
lineterminator	\n	书写器使用这个字符串结束一行
quotechar	"	这个字符串用来包围包含特殊值的字段（一个字符）
quoting	QUOTE_MINIMAL	控制前面介绍的引号行为
skipinitialspace	False	忽略字段定界符后面的空白符

```

import csv
import sys

csv.register_dialect('escaped',
                    escapechar='\\',
                    doublequote=False,
                    quoting=csv.QUOTE_NONE,
                    )
csv.register_dialect('singlequote',
                    quotechar="'",
                    quoting=csv.QUOTE_ALL,
                    )

quoting_modes = dict( (getattr(csv,n), n)
                      for n in dir(csv)
                      if n.startswith('QUOTE_')
                      )

for name in sorted(csv.list_dialects()):
    print 'Dialect: "%s"\n' % name
    dialect = csv.get_dialect(name)

    print ' delimiter = %-6r    skipinitialspace = %r' % (
        dialect.delimiter, dialect.skipinitialspace)
    print ' doublequote = %-6r    quoting          = %s' % (
        dialect.doublequote, quoting_modes[dialect.quoting])
    print ' quotechar   = %-6r    lineterminator   = %r' % (
        dialect.quotechar, dialect.lineterminator)
    print ' escapechar  = %-6r' % dialect.escapechar
    print

writer = csv.writer(sys.stdout, dialect=dialect)
writer.writerow(
    ('coll', 1, '10/01/2010',
     'Special chars: " \' %s to parse' % dialect.delimiter)

```

```
)
print
```

这个程序显示了采用多种不同的方言时相同的数据会如何显示。

```
$ python csv_dialect_variations.py
```

```
Dialect: "escaped"
```

```
delimiter    = ','          skipinitialspace = 0
doublequote  = 0            quoting           = QUOTE_NONE
quotechar    = '"'         lineterminator      = '\r\n'
escapechar   = '\\'        
```

```
coll,1,10/01/2010,Special chars: \" ' \, to parse
```

```
Dialect: "excel"
```

```
delimiter    = ','          skipinitialspace = 0
doublequote  = 1            quoting           = QUOTE_MINIMAL
quotechar    = '"'         lineterminator      = '\r\n'
escapechar   = None        
```

```
coll,1,10/01/2010,"Special chars: " ' , to parse"
```

```
Dialect: "excel-tab"
```

```
delimiter    = '\t'        skipinitialspace = 0
doublequote  = 1            quoting           = QUOTE_MINIMAL
quotechar    = '"'         lineterminator      = '\r\n'
escapechar   = None        
```

```
coll    1    10/01/2010    "Special chars: " '    to parse"
```

```
Dialect: "singlequote"
```

```
delimiter    = ','          skipinitialspace = 0
doublequote  = 1            quoting           = QUOTE_ALL
quotechar    = "'"         lineterminator      = '\r\n'
escapechar   = None        
```

```
'coll','1','10/01/2010','Special chars: " ' , to parse'
```

自动检测方言

要配置方言来解析一个输入文件，最好的办法就是提前知道正确的设置。对于方言参数未知的数据，可以用 `Sniffer` 类来做一个有根据的猜测。`sniff()` 方法取一个输入数据样本和一个可选的参数（给出可能的定界字符）。


```

import csv
from StringIO import StringIO
import textwrap

csv.register_dialect('escaped',
                    escapechar='\\',
                    doublequote=False,
                    quoting=csv.QUOTE_NONE)
csv.register_dialect('singlequote',
                    quotechar="' ",
                    quoting=csv.QUOTE_ALL)

# Generate sample data for all known dialects
samples = []
for name in sorted(csv.list_dialects()):
    buffer = StringIO()
    dialect = csv.get_dialect(name)
    writer = csv.writer(buffer, dialect=dialect)
    writer.writerow(
        ('coll', 1, '10/01/2010',
         'Special chars " \' %s to parse' % dialect.delimiter)
    )
    samples.append( (name, dialect, buffer.getvalue()) )
# Guess the dialect for a given sample, and then use the results to
# parse the data.
sniffer = csv.Sniffer()
for name, expected, sample in samples:
    print 'Dialect: "%s"\n' % name
    dialect = sniffer.sniff(sample, delimiters=',\t')
    reader = csv.reader(StringIO(sample), dialect=dialect)
    print reader.next()
    print

```

sniff() 会返回一个 Dialect 实例，其中包含用于解析数据的设置。这个结果并不总是尽善尽美，示例中的“escaped”方言可以说明这一点。

```
$ python csv_dialect_sniffer.py
```

```
Dialect: "escaped"
```

```
['coll', '1', '10/01/2010', 'Special chars \\" \' \\", ' to parse']
```

```
Dialect: "excel"
```

```
['coll', '1', '10/01/2010', 'Special chars " \' , to parse']
```

```
Dialect: "excel-tab"
```

```
['coll', '1', '10/01/2010', 'Special chars " \' \t to parse']

Dialect: "singlequote"

['coll', '1', '10/01/2010', 'Special chars " \' , to parse']
```

7.7.4 使用字段名

除了处理数据序列，csv 模块还包括一些类，可以将行作为字典来处理，从而可以对字段命名。DictReader 和 DictWriter 类将行转换为字典而不是列表。字典的键可以传入，也可以由输入的第一行推导得出（如果行包含首部）。

```
import csv
import sys

with open(sys.argv[1], 'rt') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print row
```

基于字典的读者和书写器会实现为基于序列的类的包装器，它们使用相同的方法和参数。阅读器 API 中惟一的差别是：行将作为字典返回，而不是作为列表或元组。

```
$ python csv_dictreader.py testdata.csv
```

```
{'Title 1': '1', 'Title 3': '08/18/07', 'Title 2': 'a'}
{'Title 1': '2', 'Title 3': '08/19/07', 'Title 2': 'b'}
{'Title 1': '3', 'Title 3': '08/20/07', 'Title 2': 'c'}
```

必须为 DictWriter 提供一个字段名列表，使它知道如何在输出中确定列的顺序。

```
import csv
import sys

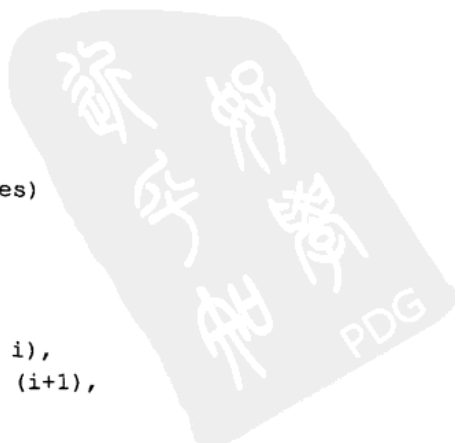
with open(sys.argv[1], 'wt') as f:

    fieldnames = ('Title 1', 'Title 2', 'Title 3')
    headers = dict( (n,n) for n in fieldnames )

    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writerow(headers)

    for i in range(3):
        writer.writerow({ 'Title 1':i+1,
                           'Title 2':chr(ord('a') + i),
                           'Title 3':'08/%02d/07' % (i+1),
                           })

print open(sys.argv[1], 'rt').read()
```



字段名并不自动写至文件，所以需要在写其他数据之前显式写出。

```
$ python csv_dictwriter.py testout.csv
```

```
Title 1,Title 2,Title 3  
1,a,08/01/07  
2,b,08/02/07  
3,c,08/03/07
```

参见：

csv (<http://docs.python.org/library/csv.html>) 这个模块的标准库文档。

PEP 305 (www.python.org/dev/peps/pep-0305) CSV 文件 API。



第 ⑧ 章

数据压缩与归档

尽管现代计算机系统的存储能力日益增长，但是所生成数据的增长是永无休止的。无损（lossless）压缩算法以压缩或解压缩数据花费的时间来换取存储数据所需要的空间，以弥补存储能力的不足。Python 为最流行的一些压缩库提供了接口，从而能交替使用不同压缩库读写文件。

zlib 和 gzip 提供了 GNU zip 库，另外 bz2 允许访问更新的 bzip2 格式。这些格式都处理数据流而不考虑输入格式，并且提供了接口可以透明地读写压缩文件。可以使用这些模块压缩单个文件或数据源。

标准库还包括一些模块来管理归档（archive）格式，将多个文件合并到一个文件，从而将其作为一个单元来管理。tarfile 读写 UNIX 磁带归档格式，这是一种老标准，但由于其灵活性，当前仍得到广泛使用。zipfile 根据 zip 格式来处理归档，这种格式因 PC 程序 PKZIP 得以普及，原先在 MS-DOS 和 Windows 下使用，不过由于其 API 的简单性以及这种格式的可移植性，现在也用于其他平台。

8.1 zlib——GNU zlib 压缩

作用：对 GNU zlib 压缩库的底层访问。

Python 版本：2.5 及以后版本

zlib 模块为 GNU 项目 zlib 压缩库中的很多函数提供了一个底层接口。

8.1.1 处理内存中数据

使用 zlib 最简单的方法要求把所有将要压缩或解压缩的数据存放在内存中：

```
import zlib
import binascii

original_data = 'This is the original text.'
print 'Original      :', len(original_data), original_data

compressed = zlib.compress(original_data)
print 'Compressed    :', len(compressed), binascii.hexlify(compressed)

decompressed = zlib.decompress(compressed)
```

```
print 'Decompressed :', len(decompressed), decompressed
```

compress() 和 decompress() 函数都取一个字符串参数，并返回一个字符串。

```
$ python zlib_memory.py
```

```
Original      : 26 This is the original text.
Compressed    : 32 789c0bc9c82c5600a2928c5485fca2ccf4c8bcc41c8592d
48a123d007f2f097e
Decompressed  : 26 This is the original text.
```

从前面的例子可以看出，对于简短的文本，一个串的压缩版本可能比未压缩的版本还要大。具体的结果取决于输入数据，观察小段文本的压缩开销很有意思。

```
import zlib
```

```
original_data = 'This is the original text.'
```

```
fmt = '%15s %15s'
```

```
print fmt % ('len(data)', 'len(compressed)')
```

```
print fmt % ('-' * 15, '-' * 15)
```

```
for i in xrange(5):
```

```
    data = original_data * i
```

```
    compressed = zlib.compress(data)
```

```
    highlight = '*' if len(data) < len(compressed) else ''
```

```
    print fmt % (len(data), len(compressed)), highlight
```

输出中的 * 突出显示了哪些行的压缩数据比未压缩版本还会占用更多内存。

```
$ python zlib_lengths.py
```

len(data)	len(compressed)
0	8 *
26	32 *
52	35
78	35
104	36

8.1.2 增量压缩与解压缩

这种内存中压缩方法存在一些缺点，主要是系统需要足够的内存，能够在内存中同时驻留未压缩和压缩版本，因此对于真实世界的用例并不实用。另一种方法是使用 Compress 和 Decompress 对象以增量方式处理数据，这样就不需要将整个数据集都放在内存中。

```
import zlib
```

```
import binascii
```

```

compressor = zlib.compressobj(1)

with open('lorem.txt', 'r') as input:
    while True:
        block = input.read(64)
        if not block:
            break
        compressed = compressor.compress(block)
        if compressed:
            print 'Compressed: %s' % binascii.hexlify(compressed)
        else:
            print 'buffering...'
    remaining = compressor.flush()
    print 'Flushed: %s' % binascii.hexlify(remaining)

```

这个例子从一个纯文本文件读取小数据块，并传至 `compress()`。压缩器维护压缩数据的一个内部缓冲区。由于压缩算法依赖于校验和以及最小块大小，所以压缩器每次接收更多输入时可能并没有准备好返回数据。如果它没有准备好一个完整的压缩块，就会返回一个空串。所有数据都已输入时，`flush()` 方法会强制压缩器结束最后一个块，并返回余下的压缩数据。

```
$ python zlib_incremental.py
```

```

Compressed: 7801
buffering...
buffering...
buffering...
buffering...
buffering...
Flushed: 55904b6ac4400c44f73e451da0f129b20c2110c85e696b8c40ddedd167ce1
f7915025a087daa9ef4be8c07e4f21c38962e834b800647435fd3b90747b2810eb9c4b
bcc13ac123bde6e4bef1c91ee40d3c6580e3ff52aad2e8cb2eb6062dad74a89ca904c
bb0f2545e0db4b1f2e01955b8c511cb2ac08967d228af1447c8ec72e40c4c714116e60
cdef171bb6c0feaa25dff1c507c2c4439ec9605b7e0ba9fc54bae39355cb89fd6ebe5
841d673c7b7bc68a46f575a312eebd220d4b32441bdc1b36ebf0aedef3d57ea4b26dd9
86dd39af57dfb05d32279de

```

8.1.3 混合内容流

如果压缩和未压缩数据混合在一起，还可以使用 `decompressobj()` 返回的 `Decompress` 类。

```

import zlib

lorem = open('lorem.txt', 'rt').read()
compressed = zlib.compress(lorem)
combined = compressed + lorem

decompressor = zlib.decompressobj()
decompressed = decompressor.decompress(combined)

```

```

decompressed_matches = decompressed == lorem
print 'Decompressed matches lorem:', decompressed_matches
unused_matches = decompressor.unused_data == lorem
print 'Unused data matches lorem :', unused_matches

```

解压缩所有数据后, `unused_data` 属性会包含未用的所有数据。

```
$ python zlib_mixed.py
```

```

Decompressed matches lorem: True
Unused data matches lorem : True

```

8.1.4 校验和

除了压缩和解压缩函数, `zlib` 还包括两个函数来计算数据的校验和, 分别是 `adler32()` 和 `crc32()`。这两个函数计算出的校验和都不能认为是密码安全的, 它们只用于数据完整性验证。

```

import zlib

data = open('lorem.txt', 'r').read()

cksum = zlib.adler32(data)
print 'Adler32: %12d' % cksum
print '          : %12d' % zlib.adler32(data, cksum)

cksum = zlib.crc32(data)
print 'CRC-32 : %12d' % cksum
print '          : %12d' % zlib.crc32(data, cksum)

```

这两个函数取相同的参数, 包括一个数据串和一个可选值, 这个值用作校验和的一个起点。函数会返回一个 32 位有符号整数值, 可以作为一个新的起点参数再传回到后续的调用, 来生成一个动态变化的校验和。

```
$ python zlib_checksums.py
```

```

Adler32:   -752715298
          :    669447099
CRC-32 :  -1256596780
          : -1424888665

```

8.1.5 压缩网络数据

下一个代码清单中的服务器使用流压缩器来响应包含文件名的请求, 它将文件的一个压缩版本写至用来与客户通信的套接字。这里人为做了一些分块, 以展示传递到 `compress()` 或 `decompress()` 的数据没有相应得到完整的压缩或未压缩输出块时, 会进行缓冲处理。

```

import zlib
import logging
import SocketServer

```



```

    )
    logger = logging.getLogger('Client')

    # Set up a server, running in a separate thread
    address = ('localhost', 0) # let the kernel assign a port
    server = SocketServer.TCPServer(address, ZlibRequestHandler)
    ip, port = server.server_address # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)
    t.start()

```

客户连接到套接字，并请求一个文件。然后循环，接收压缩数据块。由于一个块可能未包含足够的信息来完全解压缩，之前接收的剩余数据将与新数据结合，传递到解压缩器。解压缩数据时，会把它追加到一个缓冲区，处理循环结束时将与文件内容比较。

```

# Connect to the server as a client
logger.info('Contacting server on %s:%s', ip, port)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, port))
# Ask for a file
requested_file = 'lorem.txt'
logger.debug('sending filename: "%s"', requested_file)
len_sent = s.send(requested_file)
# Receive a response
buffer = StringIO()
decompressor = zlib.decompressobj()
while True:
    response = s.recv(BLOCK_SIZE)
    if not response:
        break
    logger.debug('READ "%s"', binascii.hexlify(response))

    # Include any unconsumed data when feeding the decompressor.
    to_decompress = decompressor.unconsumed_tail + response
    while to_decompress:
        decompressed = decompressor.decompress(to_decompress)
        if decompressed:
            logger.debug('DECOMPRESSED "%s"', decompressed)
            buffer.write(decompressed)
            # Look for unconsumed data due to buffer overflow
            to_decompress = decompressor.unconsumed_tail
        else:
            logger.debug('BUFFERING')
            to_decompress = None

```

```
# deal with data remaining inside the decompressor buffer
remainder = decompressor.flush()
if remainder:
    logger.debug('FLUSHED "%s"', remainder)
    buffer.write(remainder)

full_response = buffer.getvalue()
lorem = open('lorem.txt', 'rt').read()
logger.debug('response matches file contents: %s',
             full_response == lorem)

# Clean up
s.close()
server.socket.close()
```

警告：这个服务器存在明显的安全隐患。不要在开放的 Internet 中或者安全问题会产生严重影响的任何环境中运行这个程序。

```
$ python zlib_server.py
```

```
Client: Contacting server on 127.0.0.1:55085
Client: sending filename: "lorem.txt"
Server: client asked for: "lorem.txt"
Server: RAW "Lorem ipsum dolor sit amet, consectetur adipiscing elit
. Donec
"
Server: SENDING "7801"
Server: RAW "egestas, enim et consectetur ullamcorper, lectus ligula
rutrum "
Server: BUFFERING
Server: RAW "leo, a
elementum elit tortor eu quam. Duis tincidunt nisi ut ant"
Server: BUFFERING
Server: RAW "e. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque ve"
Server: BUFFERING
Server: RAW "l arcu. Vivamus
purus orci, iaculis ac, suscipit sit amet, pulvi"
Server: BUFFERING
Server: RAW "nar eu,
lacus.
"
Server: BUFFERING
Server: FLUSHING "55904b6ac4400c44f73e451da0f129b20c2110c85e696b8c40d
dedd167celf7915025a087daa9ef4be8c07e4f21c38962e834b800647435fd3b90747
b2810eb9"
```

```

Server: FLUSHING "c4bbcc13ac123bde6e4bef1c91ee40d3c6580e3ff52aad2e8c
b2eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08967d228af14
47c8ec72"
Server: FLUSHING "e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c4439ec
9605b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a312eebd220d4
b32441bd"
Server: FLUSHING "c1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32279de"
Client: READ "780155904b6ac4400c44f73e451da0f129b20c2110c85e696b8c40d
dedd167cel1f7915025a087daa9ef4be8c07e4f21c38962e834b800647435fd3b90747
b281"
Client: DECOMPRESSED "Lorem ipsum dolor sit amet, consectetur "
Client: READ "0eb9c4bbcc13ac123bde6e4bef1c91ee40d3c6580e3ff52aad2e8c
b2eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08967d228af14
47c8"
Client: DECOMPRESSED "adipiscing elit. Donec
egestas, enim et consectetur ullamcorper, lectus ligula rutrum leo,
a
elementum elit tortor eu quam. Duis ti"
Client: READ "ec72e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c4439ec
9605b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a312eebd220d4
b324"
Client: DECOMPRESSED "ncidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque vel arcu. Vivamu
s
purus orci, iacu"
Client: READ "41bdc1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32279de"
Client: DECOMPRESSED "lis ac, suscipit sit amet, pulvinar eu,
lacus.
"
Client: response matches file contents: True

```

参见:

zlib (<http://docs.python.org/library/zlib.html>) 这个模块的标准库文档。

www.zlib.net/ zlib 库的主页。

www.zlib.net/manual.html 完整的 zlib 文档。

bz2 (8.3 节) bz2 模块为 bzip2 压缩库提供了一个类似的接口。

gzip (8.2 节) gzip 模块包含 zlib 库的一个更高级的（基于文件）的接口。

8.2 gzip——读写 GNU Zip 文件

作用: 读写 gzip 文件。

Python 版本: 1.5.2 及以后版本

gzip 模块为 GNU zip 文件提供了一个类文件的接口, 它使用 zlib 来压缩和解压缩数据。

8.2.1 写压缩文件

模块级函数 `open()` 创建类文件的类 `GzipFile` 的一个实例。它提供了读写数据的常用方法。

```
import gzip
import os

outfilename = 'example.txt.gz'
with gzip.open(outfilename, 'wb') as output:
    output.write('Contents of the example file go here.\n')

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes'
os.system('file -b --mime %s' % outfilename)
```

要把数据写至一个压缩文件，需要以模式 'w' 打开文件。

```
$ python gzip_write.py
```

```
application/x-gzip; charset=binary
example.txt.gz contains 68 bytes
```

通过传入一个压缩级别 (`compresslevel`) 参数，可以使用不同的压缩量。合法值为 1~9 (包括 1 和 9)。值越小意味着压缩越快，得到的压缩程度越小。较大的值说明速度较慢，但压缩程度较大 (直到某个上限)。

```
import gzip
import os
import hashlib

def get_hash(data):
    return hashlib.md5(data).hexdigest()

data = open('lorem.txt', 'r').read() * 1024
cksum = get_hash(data)

print 'Level  Size          Checksum'
print '-----'
print 'data    %10d  %s' % (len(data), cksum)

for i in xrange(1, 10):
    filename = 'compress-level-%s.gz' % i
    with gzip.open(filename, 'wb', compresslevel=i) as output:
        output.write(data)
    size = os.stat(filename).st_size
    cksum = get_hash(open(filename, 'rb').read())
    print '%5d  %10d  %s' % (i, size, cksum)
```

输出中，中间一列的数字显示了压缩输入所生成文件的大小 (字节数)。对于这个输入数据，压缩值越高并不一定会使存储空间减少。取决于输入数据，结果会有变化。

```
$ python gzip_compresslevel.py
```

Level	Size	Checksum
data	754688	e4c0f9433723971563f08a458715119c
1	9839	3fbd996cd4d63acc70047fb62646f2ba
2	8260	427bf6183d4518bcd05611d4f114a07c
3	8221	078331b777a11572583e3fdaa120b845
4	4160	f73c478ffcba30bfe0b1d08d0f597394
5	4160	022d920880e24c1895219a31105a89c8
6	4160	45ba520d6af45e279a56bb9c67294b82
7	4160	9a834b8a2c649d4b8d509cb12cc580e2
8	4160	claaFc7d7d58cba4ef21dfce6fd1f443
9	4160	78039211f5777f9f34cf770c2eaafc6d

GzipFile 实例还包括一个 writelines() 方法，可以用来写一个字符串序列。

```
import gzip
import itertools
import os

with gzip.open('example_lines.txt.gz', 'wb') as output:
    output.writelines(
        itertools.repeat('The same line, over and over.\n', 10)
    )

os.system('gzcat example_lines.txt.gz')
```

与常规文件一样，输入行要包含一个换行符。

```
$ python gzip_writelines.py

The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
```

8.2.2 读压缩数据

要从之前压缩的文件读回数据，可以用二进制读模式 ('rb') 打开文件，这样就不会对行尾结束字符完成基于文本的转换。

```
import gzip
```

```
with gzip.open('example.txt.gz', 'rb') as input_file:
    print input_file.read()
```

这个例子读取上一节 `gzip_write.py` 所写的文件。

```
$ python gzip_read.py
```

Contents of the example file go here.

读文件时，还可以用 `seek` 定位，只读取部分数据。

```
import gzip
```

```
with gzip.open('example.txt.gz', 'rb') as input_file:
    print 'Entire file:'
    all_data = input_file.read()
    print all_data

    expected = all_data[5:15]

    # rewind to beginning
    input_file.seek(0)
    # move ahead 5 bytes
    input_file.seek(5)
    print 'Starting at position 5 for 10 bytes:'
    partial = input_file.read(10)
    print partial

    print
    print expected == partial
```

`seek()` 位置是相对未压缩数据的位置，所以调用者并不需要知道数据文件是压缩文件。

```
$ python gzip_seek.py
```

```
Entire file:
Contents of the example file go here.

Starting at position 5 for 10 bytes:
nts of the

True
```

8.2.3 处理流

`GzipFile` 类可以用来包装其他类型的数据流，使它们也能使用压缩。通过一个套接字或一个现有的（已经打开的）文件句柄传输数据时，这会很有用。还可以使用 `StringIO` 缓冲区。

```

import gzip
from cStringIO import StringIO
import binascii

uncompressed_data = 'The same line, over and over.\n' * 10
print 'UNCOMPRESSED:', len(uncompressed_data)
print uncompressed_data

buf = StringIO()
with gzip.GzipFile(mode='wb', fileobj=buf) as f:
    f.write(uncompressed_data)
compressed_data = buf.getvalue()
print 'COMPRESSED:', len(compressed_data)
print binascii.hexlify(compressed_data)

inbuffer = StringIO(compressed_data)
with gzip.GzipFile(mode='rb', fileobj=inbuffer) as f:
    reread_data = f.read(len(uncompressed_data))

print
print 'REREAD:', len(reread_data)
print reread_data

```

使用 GzipFile 而不是 zlib 的一个好处是, GzipFile 支持文件 API。不过, 重新读先前压缩的数据时, 要向 read() 显式传递一个长度。如果没有这个长度, 会导致一个 CRC 错误, 这可能是因为 StringIO 报告 EOF 之前会返回一个空串。处理压缩数据流时, 可以在数据前加一个整数作为前缀, 表示要读取的具体数据量, 或者也可以使用 zlib 中的增量解压缩 API。

```
$ python gzip_StringIO.py
```

```

UNCOMPRESSED: 300
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.

COMPRESSED: 51
1f8b08001f96f24c02ff0bc94855284ecc4d55c8c9cc4bd551c82f4b2d5248cc4
b0133f4b8424665916401d3e717802c010000

REREAD: 300

```

```
The same line, over and over.  
The same line, over and over.  
The same line, over and over.  
The same line, over and over.  
The same line, over and over.  
The same line, over and over.  
The same line, over and over.  
The same line, over and over.  
The same line, over and over.  
The same line, over and over.
```

参见:

gzip (<http://docs.python.org/library/gzip.html>) 这个模块的标准库文档。

bz2 (8.3 节) bz2 模块使用 bzip2 压缩格式。

tarfile (8.4 节) tarfile 模块包含读取压缩 tar 归档文件的内置支持。

zlib (8.1 节) zlib 模块是 gzip 压缩的一个底层接口。

zipfile (8.5 节) zipfile 模块提供了对 ZIP 归档文件的访问。

8.3 bz2——bzip2 压缩

作用: 完成 bzip2 压缩。

Python 版本: 2.3 及以后版本

bz2 模块是 bzip2 库的一个接口, 用于压缩数据以便存储或传输。它提供了 3 种 API:

- “一次性” 压缩 / 解压缩函数, 用以处理大数据块 (blob)
- 迭代式压缩 / 解压缩对象, 用来处理数据流
- 一个类文件的类, 支持像读写未压缩文件一样读写压缩文件

8.3.1 内存中一次性操作

使用 bz2 最简单的方法是将所有要压缩或解压缩的数据加载到内存中, 然后使用 `compress()` 和 `decompress()` 来完成转换。

```
import bz2  
import binascii  
  
original_data = 'This is the original text.'  
print 'Original      : %d bytes' % len(original_data)  
print original_data  
print  
compressed = bz2.compress(original_data)  
print 'Compressed    : %d bytes' % len(compressed)  
hex_version = binascii.hexlify(compressed)  
for i in xrange(len(hex_version)/40 + 1):  
    print hex_version[i*40:(i+1)*40]
```



```

print
decompressed = bz2.decompress(compressed)
print 'Decompressed : %d bytes' % len(decompressed)
print decompressed

```

压缩数据包含非 ASCII 字符，所以在打印之前需要先转换为其十六进制表示。在这些例子的输出中，重新编排了十六进制表示，使每行最多有 40 个字符。

```

$ python bz2_memory.py

Original      : 26 bytes
This is the original text.

Compressed    : 62 bytes
425a683931415926535916be35a6000002938040
01040022e59c402000314c000111e93d434da223
028cf9e73148cae0a0d6ed7f17724538509016be
35a6

Decompressed  : 26 bytes
This is the original text.

```

对于短文本，压缩版本的长度可能大大超过原版本。具体的结果取决于输入数据，不过观察压缩开销会很有意思。

```

import bz2

original_data = 'This is the original text.'

fmt = '%15s %15s'
print fmt % ('len(data)', 'len(compressed)')
print fmt % ('-' * 15, '-' * 15)
for i in xrange(5):
    data = original_data * i
    compressed = bz2.compress(data)
    print fmt % (len(data), len(compressed)),
    print '*' if len(data) < len(compressed) else ''

```

末尾是 * 字符的输出行显示了这一行压缩数据比原输入数据更长。

```
$ python bz2_lengths.py
```

len(data)	len(compressed)
0	14 *
26	62 *
52	68 *
78	70
104	72

8.3.2 增量压缩和解压缩

内存中压缩方法存在明显的缺点，对于实际用例并不实用。另一种方法是使用 `BZ2Compressor` 和 `BZ2Decompressor` 对象以增量方式处理数据，从而不必将整个数据集都放在内存中。

```
import bz2
import binascii

compressor = bz2.BZ2Compressor()

with open('lorem.txt', 'r') as input:
    while True:
        block = input.read(64)
        if not block:
            break
        compressed = compressor.compress(block)
        if compressed:
            print 'Compressed: %s' % binascii.hexlify(compressed)
        else:
            print 'buffering...'
        remaining = compressor.flush()
        print 'Flushed: %s' % binascii.hexlify(remaining)
```

这个例子从一个纯文本文件读取小数据块，将它传至 `compress()`。压缩器维护压缩数据的一个内部缓冲区。由于压缩算法取决于校验和以及最小块大小，所以压缩器每次接收更多输入时可能并没有准备好返回数据。如果它没有准备好一个完整的压缩块，会返回一个空串。所有数据都已经输入时，`flush()` 方法强制压缩器结束最后一个数据块，并返回余下的压缩数据。

```
$ python bz2_incremental.py
```

```
buffering...
buffering...
buffering...
buffering...
Flushed: 425a6839314159265359ba83a48c000014d5800010400504052fa7fe00300
0ba9112793d4ca789068698a0d1a341901a0d53f4d1119a8d4c9e812d755a67c107983
87682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6faf209c52a90aaa4d16a4a1b9c16
7a01c8d9ef32589d831e77df7a5753a398b11660e392126fc18a72a1088716cc8dedda
5d489da410748531278043d70a8a131c2b8adcd6a221bdb8c7ff76b88c1d5342ee48a7
0a12175074918
```

8.3.3 混合内容流

压缩和未压缩数据混合在一起的情况下，还可以使用 `BZ2Decompressor`。

```
import bz2
```

```

lorem = open('lorem.txt', 'rt').read()
compressed = bz2.compress(lorem)
combined = compressed + lorem

decompressor = bz2.BZ2Decompressor()
decompressed = decompressor.decompress(combined)

decompressed_matches = decompressed == lorem
print 'Decompressed matches lorem:', decompressed_matches
unused_matches = decompressor.unused_data == lorem
print 'Unused data matches lorem:', unused_matches

```

解压缩所有数据之后，unused_data 属性会包含所有未用的数据。

```
$ python bz2_mixed.py
```

```

Decompressed matches lorem: True
Unused data matches lorem : True

```

8.3.4 写压缩文件

可以用 BZ2File 读写 bzip2 压缩文件，使用常用的方法读写数据。

```

import bz2
import contextlib
import os

with contextlib.closing(bz2.BZ2File('example.bz2', 'wb')) as output:
    output.write('Contents of the example file go here.\n')

os.system('file example.bz2')

```

要把数据写入一个压缩文件，需要用模式 'w' 打开文件。

```
$ python bz2_file_write.py
```

```
example.bz2: bzip2 compressed data, block size = 900k
```

通过传入一个 compresslevel 参数，可以使用不同的压缩级别。合法值为 1~9(包括 1 和 9)。值越小意味着压缩越快，压缩程度越小。较大的值说明压缩速度较慢，但压缩程度较大(直到某个上限)。

```

import bz2
import os

data = open('lorem.txt', 'r').read() * 1024
print 'Input contains %d bytes' % len(data)

for i in xrange(1, 10):
    filename = 'compress-level-%s.bz2' % i

```

```

with bz2.BZ2File(filename, 'wb', compresslevel=i) as output:
    output.write(data)
os.system('cksum %s' % filename)

```

脚本输出中，中间一列数字显示了所生成文件的大小（字节数）。对于这个输入数据，更高的压缩值并不一定会减少存储空间。不过对于其他输入，结果可能有所不同。

```

$ python bz2_file_compresslevel.py

3018243926 8771 compress-level-1.bz2
1942389165 4949 compress-level-2.bz2
2596054176 3708 compress-level-3.bz2
1491394456 2705 compress-level-4.bz2
1425874420 2705 compress-level-5.bz2
2232840816 2574 compress-level-6.bz2
447681641 2394 compress-level-7.bz2
3699654768 1137 compress-level-8.bz2
3103658384 1137 compress-level-9.bz2
Input contains 754688 bytes

```

BZ2File 实例还包括一个 `writelines()` 方法，可以用来写一个字符串序列。

```

import bz2
import contextlib
import itertools
import os

with contextlib.closing(bz2.BZ2File('lines.bz2', 'wb')) as output:
    output.writelines(
        itertools.repeat('The same line, over and over.\n', 10),
    )

os.system('bzcata lines.bz2')

```

与写常规文件类似，这些行要以一个换行符结尾。

```

$ python bz2_file_writelines.py
_

The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.

```



8.3.5 读压缩文件

要从之前压缩的文件读回数据，需要用二进制读模式 ('rb') 打开文件，这样就不会对行末字符完成基于文本的转换。

```
import bz2
import contextlib

with contextlib.closing(bz2.BZ2File('example.bz2', 'rb')) as input:
    print input.read()
```

这个例子读取上一节 bz2_file_write.py 所写的文件。

```
$ python bz2_file_read.py
```

Contents of the example file go here.

读文件时，还有可能用 seek 定位，只读部分数据。

```
import bz2
import contextlib

with contextlib.closing(bz2.BZ2File('example.bz2', 'rb')) as input:
    print 'Entire file:'
    all_data = input.read()
    print all_data

    expected = all_data[5:15]
    # rewind to beginning
    input.seek(0)

    # move ahead 5 bytes
    input.seek(5)
    print 'Starting at position 5 for 10 bytes:'
    partial = input.read(10)
    print partial

    print
    print expected == partial
```

seek() 位置是相对未压缩数据的位置，所以调用者并不需要知道数据文件是压缩文件。这就允许将一个 BZ2File 实例传入一个原本需要常规（未压缩）文件的函数。

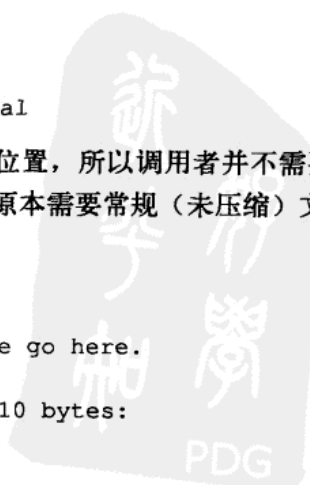
```
$ python bz2_file_seek.py
```

Entire file:

Contents of the example file go here.

Starting at position 5 for 10 bytes:

nts of the



True

8.3.6 压缩网络数据

下一个例子中的代码会响应包含文件名的请求，它将文件的一个压缩版本写至用于与客户通信的套接字。这里人为做了一些分块，以展示传递到 `compress()` 或 `decompress()` 的数据没有相应得到一个完整的压缩或未压缩输出块时，会进行缓冲处理。

```
import bz2
import logging
import SocketServer
import binascii

BLOCK_SIZE = 32

class Bz2RequestHandler(SocketServer.BaseRequestHandler):
    logger = logging.getLogger('Server')

    def handle(self):
        compressor = bz2.BZ2Compressor()

        # Find out what file the client wants
        filename = self.request.recv(1024)
        self.logger.debug('client asked for: "%s"', filename)

        # Send chunks of the file as they are compressed
        with open(filename, 'rb') as input:
            while True:
                block = input.read(BLOCK_SIZE)
                if not block:
                    break
                self.logger.debug('RAW "%s"', block)
                compressed = compressor.compress(block)
                if compressed:
                    self.logger.debug('SENDING "%s"',
                                      binascii.hexlify(compressed))
                    self.request.send(compressed)
                else:
                    self.logger.debug('BUFFERING')

        # Send any data being buffered by the compressor
        remaining = compressor.flush()
        while remaining:
            to_send = remaining[:BLOCK_SIZE]
            remaining = remaining[BLOCK_SIZE:]
            self.logger.debug('FLUSHING "%s"',
```

```

        binascii.hexlify(to_send))
    self.request.send(to_send)
    return

```

主程序在一个线程中启动一个服务器，并组合 SocketServer 和 Bz2RequestHandler。

```

if __name__ == '__main__':
    import socket
    import sys
    from cStringIO import StringIO
    import threading

    logging.basicConfig(level=logging.DEBUG,
                        format='%(name)s: %(message)s',
                        )

    # Set up a server, running in a separate thread
    address = ('localhost', 0) # let the kernel assign a port
    server = SocketServer.TCPServer(address, Bz2RequestHandler)
    ip, port = server.server_address # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)
    t.start()

    logger = logging.getLogger('Client')

    # Connect to the server
    logger.info('Contacting server on %s:%s', ip, port)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Ask for a file
    requested_file = (sys.argv[0]
                     if len(sys.argv) > 1
                     else 'lorem.txt')
    logger.debug('sending filename: "%s"', requested_file)
    len_sent = s.send(requested_file)

    # Receive a response
    buffer = StringIO()
    decompressor = bz2.BZ2Decompressor()
    while True:
        response = s.recv(BLOCK_SIZE)
        if not response:
            break
        logger.debug('READ "%s"', binascii.hexlify(response))

        # Include any unconsumed data when feeding the decompressor.

```

```
decompressed = decompressor.decompress(response)
if decompressed:
    logger.debug('DECOMPRESSED "%s"', decompressed)
    buffer.write(decompressed)
else:
    logger.debug('BUFFERING')
full_response = buffer.getvalue()
lorem = open(requested_file, 'rt').read()
logger.debug('response matches file contents: %s',
             full_response == lorem)

# Clean up
server.shutdown()
server.socket.close()
s.close()
```

然后打开一个套接字，作为客户与服务器通信，并请求文件。默认文件 `lorem.txt` 包含以下文本。

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec
egestas, enim et consectetur ullamcorper, lectus ligula rutrum leo,
a elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi.
```

警告：这个实现存在明显的安全隐患。不要在开放的 Internet 中或安全问题可能导致严重影响
的任何环境中运行这个程序。

运行 `bz2_server.py` 会生成以下结果：

```
$ python bz2_server.py
```

```
Client: Contacting server on 127.0.0.1:55091
Client: sending filename: "lorem.txt"
Server: client asked for: "lorem.txt"
Server: RAW "Lorem ipsum dolor sit amet, cons"
Server: BUFFERING
Server: RAW "ectetuer adipiscing elit. Donec
"
Server: BUFFERING
Server: RAW "egestas, enim et consectetur ul"
Server: BUFFERING
Server: RAW "lamcorper, lectus ligula rutrum "
Server: BUFFERING
Server: RAW "leo,
a elementum elit tortor eu "
Server: BUFFERING
Server: RAW "quam. Duis tincidunt nisi ut ant"
```




```

Server: BUFFERING
Server: RAW "e. Nulla
facilisi.
"
Server: BUFFERING
Server: FLUSHING "425a6839314159265359ba83a48c000014d580001040050405
2fa7fe003000ba"
Server: FLUSHING "9112793d4ca789068698a0d1a341901a0d53f4d1119a8d4c9e
812d755a67c107"
Server: FLUSHING "98387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6faf20
9c52a90aaa4d16"
Server: FLUSHING "a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b11660e
392126fc18a72a"
Server: FLUSHING "1088716cc8dedda5d489da410748531278043d70a8a131c2b8
adcd6a221bdb8c"
Server: FLUSHING "7ff76b88c1d5342ee48a70a12175074918"
Client: READ "425a6839314159265359ba83a48c000014d5800010400504052fa7
fe003000ba"
Client: BUFFERING
Client: READ "9112793d4ca789068698a0d1a341901a0d53f4d1119a8d4c9e812d
755a67c107"
Client: BUFFERING
Client: READ "98387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6faf209c52
a90aaa4d16"
Client: BUFFERING
Client: READ "a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b11660e3921
26fc18a72a"
Client: BUFFERING
Client: READ "1088716cc8dedda5d489da410748531278043d70a8a131c2b8adcd
6a221bdb8c"
Client: BUFFERING
Client: READ "7ff76b88c1d5342ee48a70a12175074918"
Client: DECOMPRESSED "Lorem ipsum dolor sit amet, consectetur adipi
scing elit. Donec
egestas, enim et consectetur ullamcorper, lectus ligula rutrum leo,
a elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi.
"
Client: response matches file contents: True

```

参见:

bz2 (<http://docs.python.org/library/bz2.html>) 这个模块的标准库文档。

bzip2.org (www.bzip.org/) bzip2 的主页。

gzip (8.2 节) GNU zip 压缩文件的一个类文件的接口。

zlib (8.1 节) zlib 模块, 用来完成 GNU zip 压缩。

8.4 tarfile——Tar 归档访问

作用：读写 tar 归档文件。

Python 版本：2.3 及以后版本

tarfile 模块提供了对 UNIX tar 归档文件（包括压缩文件）的读写访问。除 POSIX 标准外，还支持多个 GNU tar 扩展。另外还能处理一些 UNIX 特殊文件类型（如硬 / 软链接）以及设备节点。

注意：尽管 tarfile 实现了一种 UNIX 格式，也可以在 Microsoft Windows 下用它创建和读取 tar 归档文件。

8.4.1 测试 Tar 文件

is_tarfile() 函数返回一个布尔值，指示作为参数传入的文件名是否指向一个合法的 tar 归档文件。

```
import tarfile

for filename in [ 'README.txt', 'example.tar',
                  'bad_example.tar', 'notthere.tar' ]:
    try:
        print '%15s %s' % (filename, tarfile.is_tarfile(filename))
    except IOError, err:
        print '%15s %s' % (filename, err)
```

如果这个文件不存在，is_tarfile() 会产生一个 IOError。

```
$ python tarfile_is_tarfile.py
```

```
    README.txt  False
    example.tar  True
    bad_example.tar  False
    notthere.tar  [Errno 2] No such file or directory: 'notthere.tar'
```

8.4.2 从归档文件读取元数据

可以使用 TarFile 类直接处理 tar 归档文件。它支持很多方法来读取现有归档文件的有关数据，还可以添加另外的文件来修改归档。

要读取一个现有归档文件中的文件名，可以使用 getnames()。

```
import tarfile
from contextlib import closing

with closing(tarfile.open('example.tar', 'r')) as t:
    print t.getnames()
```

这个函数的返回值是一个字符串列表，包含归档内容中的文件名。

```
$ python tarfile_getnames.py
```

```
['README.txt', '__init__.py']
```

除文件名外，还可以作为 TarInfo 对象实例得到归档成员的元数据。

```
import tarfile
import time
from contextlib import closing

with closing(tarfile.open('example.tar', 'r')) as t:
    for member_info in t.getmembers():
        print member_info.name
        print '\tModified:\t', time.ctime(member_info.mtime)
        print '\tMode      :\t', oct(member_info.mode)
        print '\tType       :\t', member_info.type
        print '\tSize        :\t', member_info.size, 'bytes'
        print
```

可以通过 getmembers() 和 getmember() 加载元数据。

```
$ python tarfile_getmembers.py
README.txt
    Modified:      Sun Nov 28 13:30:14 2010
    Mode          :      0644
    Type          :      0
    Size          :      75 bytes
```

```
__init__.py
    Modified:      Sun Nov 14 09:39:38 2010
    Mode          :      0644
    Type          :      0
    Size          :      22 bytes
```

如果提前已经知道归档成员名，可以用 getmember() 获取其 TarInfo 对象。

```
import tarfile
import time
from contextlib import closing

with closing(tarfile.open('example.tar', 'r')) as t:
    for filename in ['README.txt', 'notthere.txt']:
        try:
            info = t.getmember(filename)
        except KeyError:
            print 'ERROR: Did not find %s in tar archive' % filename
        else:
            print '%s is %d bytes' % (info.name, info.size)
```

如果归档成员不存在，`getmember()` 会产生一个 `KeyError`。

```
$ python tarfile_getmember.py
```

```
README.txt is 75 bytes
```

```
ERROR: Did not find notthere.txt in tar archive
```

8.4.3 从归档抽取文件

要在程序中访问一个归档成员的数据，可以使用 `extractfile()` 方法，并传入这个成员名。

```
import tarfile
from contextlib import closing
with closing(tarfile.open('example.tar', 'r')) as t:
    for filename in [ 'README.txt', 'notthere.txt' ]:
        try:
            f = t.extractfile(filename)
        except KeyError:
            print 'ERROR: Did not find %s in tar archive' % filename
        else:
            print filename,
            print f.read()
```

返回值是一个类文件的对象，可以从这个对象读取归档成员的内容。

```
$ python tarfile_extractfile.py
```

```
README.txt :
```

```
The examples for the tarfile module use this file and example.tar as
data.
```

```
ERROR: Did not find notthere.txt in tar archive
```

要解开归档，将文件写至文件系统，可以使用 `extract()` 或 `extractall()`。

```
import tarfile
import os
from contextlib import closing

os.mkdir('outdir')
with closing(tarfile.open('example.tar', 'r')) as t:
    t.extract('README.txt', 'outdir')
print os.listdir('outdir')
```

归档成员会从归档中读出，写至文件系统，从参数中指定的目录开始。

```
$ python tarfile_extract.py
```

```
['README.txt']
```

标准库文档中有一个说明，指出 `extractall()` 比 `extract()` 更安全，特别是处理流数据，因为

对于流数据，无法回转输入返回去读之前的部分。大多数情况下都应该使用 `extractall()`。

```
import tarfile
import os
from contextlib import closing

os.mkdir('outdir')
with closing(tarfile.open('example.tar', 'r')) as t:
    t.extractall('outdir')
print os.listdir('outdir')
```

使用 `extractall()` 时，第一个参数是一个目录名，文件将写至这个目录。

```
$ python tarfile_extractall.py
```

```
['__init__.py', 'README.txt']
```

要从归档抽取特定的文件，可以把这个文件名或 `TarInfo` 元数据容器传递到 `extractall()`。

```
import tarfile
import os
from contextlib import closing

os.mkdir('outdir')
with closing(tarfile.open('example.tar', 'r')) as t:
    t.extractall('outdir',
                 members=[t.getmember('README.txt')],
                 )
print os.listdir('outdir')
```

如果提供了一个成员列表，则只抽取指定的文件。

```
$ python tarfile_extractall_members.py
```

```
['README.txt']
```

8.4.4 创建新归档

要创建一个新归档，需要用模式 'w' 打开 `TarFile`。

```
import tarfile
from contextlib import closing

print 'creating archive'
with closing(tarfile.open('tarfile_add.tar', mode='w')) as out:
    print 'adding README.txt'
    out.add('README.txt')

print
print 'Contents:'
with closing(tarfile.open('tarfile_add.tar', mode='r')) as t:
```



```

for member_info in t.getmembers():
    print member_info.name

```

现有的文件会删除，重建一个新的归档。要添加文件，可以使用 add() 方法。

```
$ python tarfile_add.py
```

```

creating archive
adding README.txt

```

```

Contents:
README.txt

```

8.4.5 使用候选归档成员名

向归档添加一个文件时，可以不用原始文件名而是用另外一个名字，由候选的 arcname 构造一个 TarInfo 对象，并把它传至 addfile()。

```

import tarfile
from contextlib import closing

print 'creating archive'
with closing(tarfile.open('tarfile_addfile.tar', mode='w')) as out:
    print 'adding README.txt as RENAMED.txt'
    info = out.gettarinfo('README.txt', arcname='RENAMED.txt')
    out.addfile(info)

print
print 'Contents:'
with closing(tarfile.open('tarfile_addfile.tar', mode='r')) as t:
    for member_info in t.getmembers():
        print member_info.name

```

归档只包含修改后的文件名。

```
$ python tarfile_addfile.py
```

```

creating archive
adding README.txt as RENAMED.txt

```

```

Contents:
RENAMED.txt

```

8.4.6 从非文件源写数据

有时，可能需要将数据从内存直接写至一个归档。并不是将数据先写入一个文件，然后再把这个文件添加到归档，这时可以使用 addfile() 从一个打开的类文件句柄添加数据：

```

import tarfile
from cStringIO import StringIO
from contextlib import closing

```

```

data = 'This is the data to write to the archive.'

with closing(tarfile.open('addfile_string.tar', mode='w')) as out:
    info = tarfile.TarInfo('made_up_file.txt')
    info.size = len(data)
    out.addfile(info, StringIO(data))

print 'Contents:'
with closing(tarfile.open('addfile_string.tar', mode='r')) as t:
    for member_info in t.getmembers():
        print member_info.name
        f = t.extractfile(member_info)
        print f.read()

```

首先构造一个 TarInfo 对象，可以为归档成员指定所需的任何名字。设置大小之后，以一个 StringIO 缓冲区作为数据源，可以使用 addfile() 把数据写至归档。

```
$ python tarfile_addfile_string.py
```

```

Contents:
made_up_file.txt
This is the data to write to the archive.

```

8.4.7 追加到归档

除了创建新归档，还可以使用模式 'a' 追加到一个现有的文件。

```

import tarfile
from contextlib import closing

print 'creating archive'
with closing(tarfile.open('tarfile_append.tar', mode='w')) as out:
    out.add('README.txt')

print 'contents:',
with closing(tarfile.open('tarfile_append.tar', mode='r')) as t:
    print [m.name for m in t.getmembers()]

print 'adding index.rst'
with closing(tarfile.open('tarfile_append.tar', mode='a')) as out:
    out.add('index.rst')

print 'contents:',
with closing(tarfile.open('tarfile_append.tar', mode='r')) as t:
    print [m.name for m in t.getmembers()]

```

最后得到的归档将包含两个成员。

```
$ python tarfile_append.py
```

```

creating archive
contents: ['README.txt']
adding index.rst
contents: ['README.txt', 'index.rst']

```

8.4.8 处理压缩归档

除了常规的 tar 归档文件，tarfile 模块还可以处理通过 gzip 或 bzip2 协议压缩的归档。要打开一个压缩归档，可以修改传入 open() 的模式串，取决于所需的压缩方法，要在模式串中包含 ":gz" 或 ":bz2"。

```

import tarfile
import os

fmt = '%-30s %-10s'
print fmt % ('FILENAME', 'SIZE')
print fmt % ('README.txt', os.stat('README.txt').st_size)

for filename, write_mode in [
    ('tarfile_compression.tar', 'w'),
    ('tarfile_compression.tar.gz', 'w:gz'),
    ('tarfile_compression.tar.bz2', 'w:bz2'),
]:
    out = tarfile.open(filename, mode=write_mode)
    try:
        out.add('README.txt')
    finally:
        out.close()

    print fmt % (filename, os.stat(filename).st_size),
    print [m.name
            for m in tarfile.open(filename, 'r:*').getmembers()]

```

打开一个现有的归档读取数据时，可以指定 "r:*" 让 tarfile 自动确定要使用的压缩方法。

```
$ python tarfile_compression.py
```

FILENAME	SIZE	
README.txt	75	
tarfile_compression.tar	10240	['README.txt']
tarfile_compression.tar.gz	212	['README.txt']
tarfile_compression.tar.bz2	187	['README.txt']

参见：

tarfile (<http://docs.python.org/library/tarfile.html>) 这个模块的标准库文档。

GNU tar manual (www.gnu.org/software/tar/manual/html_node/Standard.html) tar 格式的文

档（包括扩展）。

bz2 (8.3 节) bz2 压缩。

contextlib (3.4 节) contextlib 模块包括 closing(), 用于在 with 语句中管理文件句柄。

gzip (8.2 节) GNU zip 压缩。

zipfile (8.5 节) 对 ZIP 归档完成类似的访问。

8.5 zipfile——ZIP 归档访问

作用：读写 ZIP 归档文件。

Python 版本：1.6 及以后版本

zipfile 模块可以用来管理 ZIP 归档文件，这种格式因 PC 程序 PKZIP 得到普及。

8.5.1 测试 ZIP 文件

is_zipfile() 函数返回一个布尔值，指示作为参数传入的文件名是否指向一个合法的 ZIP 归档。

```
import zipfile

for filename in [ 'README.txt', 'example.zip',
                  'bad_example.zip', 'notthere.zip' ]:
    print '%15s %s' % (filename, zipfile.is_zipfile(filename))
```

如果这个文件不存在，则 is_zipfile() 返回 False。

```
$ python zipfile_is_zipfile.py
```

```
    README.txt  False
    example.zip  True
    bad_example.zip  False
    notthere.zip  False
```

8.5.2 从归档读取元数据

使用 ZipFile 类可以直接处理一个 ZIP 归档。它支持一些方法来读取现有归档的有关数据，还可以增加额外的文件从而修改归档。

```
import zipfile

with zipfile.ZipFile('example.zip', 'r') as zf:
    print zf.namelist()
```

namelist() 方法返回一个现有归档中的文件名。

```
$ python zipfile_namelist.py
```

```
['README.txt']
```

不过，这个文件名列表只是从归档得到的信息的一部分。要访问有关 ZIP 内容的所有元数据，可以使用 `infolist()` 或 `getinfo()` 方法。

```
import datetime
import zipfile

def print_info(archive_name):
    with zipfile.ZipFile(archive_name) as zf:
        for info in zf.infolist():
            print info.filename
            print '\tComment      :', info.comment
            mod_date = datetime.datetime(*info.date_time)
            print '\tModified      :', mod_date
            if info.create_system == 0:
                system = 'Windows'
            elif info.create_system == 3:
                system = 'Unix'
            else:
                system = 'UNKNOWN'
            print '\tSystem        :', system
            print '\tZIP version :', info.create_version
            print '\tCompressed   :', info.compress_size, 'bytes'
            print '\tUncompressed:', info.file_size, 'bytes'
            print

if __name__ == '__main__':
    print_info('example.zip')
```

除了这里打印的字段，元数据还包括另外一些字段，但是要把这些值解释为有用的信息，需要仔细阅读 ZIP 文件规范的 PKZIP 应用说明。

```
$ python zipfile_infolist.py
```

```
README.txt
Comment      :
Modified     : 2010-11-15 06:48:02
System      : Unix
ZIP version : 30
Compressed  : 65 bytes
Uncompressed: 76 bytes
```

如果提前已经知道归档成员名，可以利用 `getinfo()` 直接获取其 `ZipInfo` 对象。

```
import zipfile

with zipfile.ZipFile('example.zip') as zf:
    for filename in ['README.txt', 'notthere.txt']:
        try:
            info = zf.getinfo(filename)
```

```

except KeyError:
    print 'ERROR: Did not find %s in zip file' % filename
else:
    print '%s is %d bytes' % (info.filename, info.file_size)

```

如果归档成员不存在, getinfo() 会产生一个 KeyError。

```
$ python zipfile_getinfo.py
```

```

README.txt is 76 bytes
ERROR: Did not find notthere.txt in zip file

```

8.5.3 从归档抽取归档文件

要从一个归档成员访问数据, 可以使用 read() 方法, 并传入该成员名。

```

import zipfile

with zipfile.ZipFile('example.zip') as zf:
    for filename in [ 'README.txt', 'notthere.txt' ]:
        try:
            data = zf.read(filename)
        except KeyError:
            print 'ERROR: Did not find %s in zip file' % filename
        else:
            print filename, ':'
            print data
    print

```

如果必要, 数据会自动解压缩。

```
$ python zipfile_read.py
```

```

README.txt :
The examples for the zipfile module use
this file and example.zip as data.

```

```
ERROR: Did not find notthere.txt in zip file
```

8.5.4 创建新归档

要创建一个新归档, 需要用模式 'w' 实例化 ZipFile。所有现有的文件会被删除, 而开始创建一个新的归档。要增加文件, 可以使用 write() 方法。

```

from zipfile_infolist import print_info
import zipfile

print 'creating archive'
with zipfile.ZipFile('write.zip', mode='w') as zf:
    print 'adding README.txt'

```

```
zf.write('README.txt')
```

```
print
print_info('write.zip')
```

默认情况下，归档的内容不会压缩。

```
$ python zipfile_write.py
```

```
creating archive
adding README.txt
```

```
README.txt
      Comment      :
      Modified     : 2010-11-15 06:48:00
      System       : Unix
      ZIP version  : 20
      Compressed   : 76 bytes
      Uncompressed : 76 bytes
```

要想增加压缩，需要有 `zlib` 模块。如果 `zlib` 可用，可以使用 `zipfile.ZIP_DEFLATED` 设置单个文件的压缩模式，或者将归档作为整体设置其压缩模式。默认的压缩模式是 `zipfile.ZIP_STORED`，它会把输入数据增加到归档而不压缩。

```
from zipfile_infolist import print_info
import zipfile
try:
    import zlib
    compression = zipfile.ZIP_DEFLATED
except:
    compression = zipfile.ZIP_STORED

modes = { zipfile.ZIP_DEFLATED: 'deflated',
           zipfile.ZIP_STORED:  'stored',
           }

print 'creating archive'
with zipfile.ZipFile('write_compression.zip', mode='w') as zf:
    mode_name = modes[compression]
    print 'adding README.txt with compression mode', mode_name
    zf.write('README.txt', compress_type=compression)

print
print_info('write_compression.zip')
```

这一次归档成员会压缩。

```
$ python zipfile_write_compression.py
```

```

creating archive
adding README.txt with compression mode deflated

README.txt
  Comment      :
  Modified     : 2010-11-15 06:48:00
  System       : Unix
  ZIP version  : 20
  Compressed   : 65 bytes
  Uncompressed: 76 bytes

```

8.5.5 使用候选归档成员名

为文档增加文件时，可以向 `write()` 传入一个 `arcname` 值，这样可以使用另一个名字而非原始文件名。

```

from zipfile_infolist import print_info
import zipfile

with zipfile.ZipFile('write_arcname.zip', mode='w') as zf:
    zf.write('README.txt', arcname='NOT_README.txt')

print_info('write_arcname.zip')

归档中不会再出现原始文件名。
$ python zipfile_write_arcname.py

```

```

NOT_README.txt
  Comment      :
  Modified     : 2010-11-15 06:48:00
  System       : Unix
  ZIP version  : 20
  Compressed   : 76 bytes
  Uncompressed: 76 bytes

```

8.5.6 从非文件源写数据

有时，可能需要使用其他来源的数据写一个 ZIP 归档（而非来自一个现有文件）。并不是将数据先写入一个文件，然后再把这个文件添加到 ZIP 归档，可以使用 `writestr()` 直接向归档写入一个字节串。

```

from zipfile_infolist import print_info
import zipfile

msg = 'This data did not exist in a file.'
with zipfile.ZipFile('writestr.zip',
    mode='w',
    compression=zipfile.ZIP_DEFLATED,

```

```

        ) as zf:
            zf.writestr('from_string.txt', msg)

    print_info('writestr.zip')

    with zipfile.ZipFile('writestr.zip', 'r') as zf:
        print zf.read('from_string.txt')

```

在这里，要利用 ZipFile 的 `compress_type` 参数压缩数据，因为 `writestr()` 不接受参数来指定压缩。

```
$ python zipfile_writestr.py
```

```

from_string.txt
      Comment      :
      Modified     : 2010-11-28 13:48:46
      System       : Unix
      ZIP version  : 20
      Compressed   : 36 bytes
      Uncompressed: 34 bytes

```

```
This data did not exist in a file.
```

8.5.7 利用 ZipInfo 实例写

正常情况下，文件或串添加到归档时会计算修改日期。可以向 `writestr()` 传递一个 `ZipInfo` 实例，来定义修改日期和其他元数据。

```

import time
import zipfile
from zipfile.infolist import print_info
msg = 'This data did not exist in a file.'

with zipfile.ZipFile('writestr_zipinfo.zip',
                    mode='w',
                    ) as zf:
    info = zipfile.ZipInfo('from_string.txt',
                          date_time=time.localtime(time.time()),
                          )
    info.compress_type=zipfile.ZIP_DEFLATED
    info.comment='Remarks go here'
    info.create_system=0
    zf.writestr(info, msg)

print_info('writestr_zipinfo.zip')

```

在这个例子中，修改时间设置为当前时间，并对数据进行了压缩，另外 `create_system` 使用了一个 `false` 值。还可以为这个新文件关联一个简单的注释。

```
$ python zipfile_writestr_zipinfo.py
```

```

from_string.txt
    Comment      : Remarks go here
    Modified     : 2010-11-28 13:48:46
    System       : Windows
    ZIP version  : 20
    Compressed   : 36 bytes
    Uncompressed: 34 bytes

```

8.5.8 追加到文件

除了创建新归档，还可以追加到一个现有的归档，或者将一个归档添加到一个现有文件的末尾（如为一个自解压归档添加一个 .exe 文件）。要打开一个文件来完成追加，可以使用模式 'a'。

```

from zipfile import ZipFile, print_info
import zipfile

print 'creating archive'
with ZipFile('append.zip', mode='w') as zf:
    zf.write('README.txt')
print
print_info('append.zip')

print 'appending to the archive'
with ZipFile('append.zip', mode='a') as zf:
    zf.write('README.txt', arcname='README2.txt')

print
print_info('append.zip')

```

最后得到的归档将包含两个成员：

```
$ python zipfile_append.py
```

```
creating archive
```

```

README.txt
    Comment      :
    Modified     : 2010-11-15 06:48:00
    System       : Unix
    ZIP version  : 20
    Compressed   : 76 bytes
    Uncompressed: 76 bytes

```

```
appending to the archive
```

```

README.txt
    Comment      :
    Modified     : 2010-11-15 06:48:00

```



```

System      : Unix
ZIP version : 20
Compressed  : 76 bytes
Uncompressed: 76 bytes

```

```

README2.txt
Comment      :
Modified     : 2010-11-15 06:48:00
System      : Unix
ZIP version  : 20
Compressed   : 76 bytes
Uncompressed : 76 bytes

```

8.5.9 Python ZIP 归档

如果归档出现在 `sys.path` 中, Python 可以使用 `zipimport` 从这些 ZIP 归档导入模块。 `PyZipFile` 类可以用来构造一个适用于这种方式的模块。另一个方法 `writepy()` 会告诉 `PyZipFile` 扫描一个目录来查找 `.py` 文件, 并把相应的 `.pyo` 或 `.pyc` 文件添加到归档。如果这两种编译形式都不存在, 则创建并添加一个 `.pyc` 文件。

```

import sys
import zipfile

if __name__ == '__main__':
    with zipfile.PyZipFile('pyzipfile.zip', mode='w') as zf:
        zf.debug = 3
        print 'Adding python files'
        zf.writepy('.')
        for name in zf.namelist():
            print name

    print
    sys.path.insert(0, 'pyzipfile.zip')
    import zipfile_pyzipfile
    print 'Imported from:', zipfile_pyzipfile.__file__

```

将 `PyZipFile` 的 `debug` 属性设置为 3, 将启用 `verbose` 调试, 会在编译所找到的各个 `.py` 文件时生成输出。

```
$ python zipfile_pyzipfile.py
```

```

Adding python files
Adding package in . as .
Adding ./__init__.pyc
Adding ./zipfile_append.pyc
Adding ./zipfile_getinfo.pyc
Adding ./zipfile_infolist.pyc

```



```
Compiling ./zipfile_is_zipfile.py
Adding ./zipfile_is_zipfile.pyc
Adding ./zipfile_namelist.pyc
Adding ./zipfile_printdir.pyc
Adding ./zipfile_pyzipfile.pyc
Adding ./zipfile_read.pyc
Adding ./zipfile_write.pyc
Adding ./zipfile_write_arcname.pyc
Adding ./zipfile_write_compression.pyc
Adding ./zipfile_writestr.pyc
Adding ./zipfile_writestr_zipinfo.pyc
__init__.pyc
zipfile_append.pyc
zipfile_getinfo.pyc
zipfile_infolist.pyc
zipfile_is_zipfile.pyc
zipfile_namelist.pyc
zipfile_printdir.pyc
zipfile_pyzipfile.pyc
zipfile_read.pyc
zipfile_write.pyc
zipfile_write_arcname.pyc
zipfile_write_compression.pyc
zipfile_writestr.pyc
zipfile_writestr_zipinfo.pyc

Imported from: pyzipfile.zip/zipfile_pyzipfile.pyc
```

8.5.10 限制

zipfile 模块不支持有追加注释或多磁盘归档的 ZIP 文件，但它确实支持使用 ZIP64 扩展的超过 4GB 的 ZIP 文件。

参见：

tarfile (8.4 节) 读写 tar 归档文件。

zipfile (<http://docs.python.org/library/zipfile.html>) 这个模块的标准库文档。

zipimport (19.2 节) 从 ZIP 归档导入 Python 模块。

zlib (8.1 节) ZIP 压缩库。

PKZIP Application Note (www.pkware.com/documents/casestudies/APPNOTE.TXT) ZIP 归档格式的官方规范。

第 ⑨ 章

加 密

加密可以保护消息安全，从而能验证其正确性，并保护其不被截获。Python 的加密支持包括 hashlib 和 hmac。hashlib 使用标准算法生成消息内容的签名，如 MD5 和 SHA，hmac 则用于验证一个消息在传输过程中未被修改。

9.1 hashlib——密码散列

作用：生成密码散列和消息摘要。

Python 版本：2.5 及以后版本

hashlib 模块的出现使得不再使用单独的 md5 和 sha 模块，并使它们有了一致的 API。要使用一个特定的散列算法，可以用适当的构造函数创建一个散列对象。有了散列对象，不论使用哪一个具体的算法，对象都将使用相同的 API。

由于 hashlib 得到了 OpenSSL 的“支持”，OpenSSL 库提供的所有算法都可用，包括：

- md5
- sha1
- sha224
- sha256
- sha384
- sha512

9.1.1 示例数据

这一节中的所有例子都使用相同的示例数据：

```
import hashlib
```

```
lorem = '''Lorem ipsum dolor sit amet, consectetur adipiscing elit,  
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut  
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi  
ut aliquip ex ea commodo consequat. Duis aute irure dolor in  
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla  
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in  
culpa qui officia deserunt mollit anim id est laborum.'''
```

9.1.2 MD5 示例

要为一个数据块（在这里就是一个 ASCII 串）计算 MD5 散列或摘要，首先要创建散列对象，然后添加数据，并调用 `digest()` 或 `hexdigest()`。

```
import hashlib

from hashlib_data import lorem

h = hashlib.md5()
h.update(lorem)
print h.hexdigest()
```

这个例子使用了 `hexdigest()` 方法而不是 `digest()`，因为要对输出格式化，以便清楚地打印。如果可以接受二进制摘要值，则使用 `digest()`。

```
$ python hashlib_md5.py

1426f365574592350315090e295ac273
```

9.1.3 SHA1 示例

SHA1 摘要也采用同样的方式计算。

```
import hashlib

from hashlib_data import lorem

h = hashlib.sha1()
h.update(lorem)
print h.hexdigest()
```

这个例子中的摘要值有所不同，因为 MD5 和 SHA1 算法不同。

```
$ python hashlib_sha1.py

8173396ba8a560b89a3f3e2fcc024b044bc83d0a
```

9.1.4 按名创建散列

有些情况下，用一个字符串按名指示算法比直接使用构造函数更为方便。例如，如果能够把散列类型存储在一个配置文件中，这会很有用。在这些情况下，可以使用 `new()` 来创建一个散列计算器。

```
import hashlib
import sys

try:
    hash_name = sys.argv[1]
except IndexError:
```

```

    print 'Specify the hash name as the first argument.'
else:
    try:
        data = sys.argv[2]
    except IndexError:
        from hashlib_data import lorem as data

    h = hashlib.new(hash_name)
    h.update(data)
    print h.hexdigest()

```

运行时可以提供不同的参数:

```
$ python hashlib_new.py sha1
```

```
8173396ba8a560b89a3f3e2fcc024b044bc83d0a
```

```
$ python hashlib_new.py sha256
```

```
dca37495608c68ec23bbb54ab9675bf0152db63e5a51ab1061dc9982b843e767
```

```
$ python hashlib_new.py sha512
```

```
0e3d4bc1cbc117382fa077b147a7ff6363f6cbc7508877460f978a566a0adb6dbb4c8
```

```
b89f56514da98eb94d7135e1b7ad7fc4a2d747c02af67fcd4e571bd54de
```

```
$ python hashlib_new.py md5
```

```
1426f365574592350315090e295ac273
```

9.1.5 增量更新

散列计算器的 `update()` 方法可以反复调用。每次调用时, 都会根据提供的附加文本更新摘要。增量更新比将整个文件读入内存更高效, 而且能生成相同的结果。

```

import hashlib

from hashlib_data import lorem

h = hashlib.md5()
h.update(lorem)
all_at_once = h.hexdigest()

def chunkize(size, text):
    "Return parts of the text in size-based increments."
    start = 0
    while start < len(text):
        chunk = text[start:start+size]
        yield chunk
        start += size

```



```

    return

h = hashlib.md5()
for chunk in chunkize(64, lorem):
    h.update(chunk)
line_by_line = h.hexdigest()
print 'All at once :', all_at_once
print 'Line by line:', line_by_line
print 'Same          :', (all_at_once == line_by_line)

```

这个例子展示了在读取或生成数据时如何以增量方式更新一个摘要。

```
$ python hashlib_update.py
```

```

All at once : 1426f365574592350315090e295ac273
Line by line: 1426f365574592350315090e295ac273
Same          : True

```

参见:

hashlib (<http://docs.python.org/library/hashlib.html>) 这个模块的标准库文档。

Voidspace: IronPython and hashlib(www.voidspace.org.uk/python/weblog/arch_d7_2006_10_07.shtml#e497) 用于 IronPython 的 hashlib 的一个包装器。

hmac (9.2 节) hmac 模块。

OpenSSL (<http://www.openssl.org/>) 一个开源加密工具包。

9.2 hmac——密码消息签名与验证

作用: hmac 模块实现密钥散列来完成消息认证, 如 RFC 2104 所述。

Python 版本: 2.2 及以后版本

HMAC 算法可以用于验证信息的完整性, 这些信息可能在应用之间传递, 或者存储在一个可能有安全威胁的位置上。基本思想是生成实际数据的一个密码散列, 并提供一个共享的秘密密钥。然后使用得到的散列检查所传输或存储的消息, 确定一个信任级别, 而不再传输秘密密钥。

警告: 免责声明 本书并不提供专家安全建议。要全面了解 HMAC 的详细信息, 请查看 RFC 2104 (<http://tools.ietf.org/html/rfc2104.html>)。

9.2.1 消息签名

new() 函数会创建一个新对象来计算消息签名。下面这个例子使用了默认的 MD5 散列算法。

```
import hmac
```

```
digest_maker = hmac.new('secret-shared-key-goes-here')
```

```

with open('lorem.txt', 'rb') as f:
    while True:
        block = f.read(1024)
        if not block:
            break
        digest_maker.update(block)

digest = digest_maker.hexdigest()
print digest

```

运行这个代码时，会读取一个数据文件，为它计算一个 HMAC 签名。

```
$ python hmac_simple.py
```

```
4bcb287e284f8c21e87e14ba2dc40b16
```

9.2.2 SHA 与 MD5

尽管 hmac 的默认密码算法是 MD5，但这并不是最安全的方法。MD5 散列有一些缺点，如冲突（两个不同的消息生成相同的散列）。一般认为 SHA1 算法更健壮，推荐使用。

```

import hmac
import hashlib

digest_maker = hmac.new('secret-shared-key-goes-here',
                        '',
                        hashlib.sha1)

with open('hmac_sha.py', 'rb') as f:
    while True:
        block = f.read(1024)
        if not block:
            break
        digest_maker.update(block)

digest = digest_maker.hexdigest()
print digest

```

`new()` 函数有 3 个参数。第 1 个参数是秘密密钥，这个密钥会在通信双方之间共享，使两端都可以使用相同的值。第 2 个值是一个初始消息。如果需要认证的消息内容很小，如 1 个时间戳或 1 个 HTTP POST，则把消息的整个主体都传递到 `new()` 而不是使用 `update()` 方法。最后 1 个参数是要使用的摘要模块。默认为 `hashlib.md5`。这个例子将算法替换为 `hashlib.sha1`。

```
$ python hmac_sha.py
```

```
b9e8c6737883a9d3a258a0b5090559b7e8e2efcb
```

9.2.3 二进制摘要

前面的例子使用 `hexdigest()` 方法来生成可打印的摘要。`hexdigest` 是 `digest()` 方法计算出的值的一个不同表示，这是一个二进制值，可以包括不可打印的字符或非 ASCII 字符（包括 NUL）。有些 Web 服务（Google checkout、Amazon S3）会使用 Base64 编码版本的二进制摘要而不是 `hexdigest`。

```
import base64
import hmac
import hashlib

with open('lorem.txt', 'rb') as f:
    body = f.read()

hash = hmac.new('secret-shared-key-goes-here', body, hashlib.sha1)
digest = hash.digest()
print base64.encodestring(digest)
```

Base64 编码串以一个换行符结束，将这个串嵌在 http 首部或其他格式敏感的上下文中时通常需要去除这个换行符。

```
$ python hmac_base64.py

0lW2DoXHGJEKGU0aE9fOwSVE/o4=
```

9.2.4 消息签名的应用

对于所有公共网络服务，或者在安全性要求很高的位置存储数据，都应当使用 HMAC 认证。例如，通过一个管道或套接字发送数据时，应当对数据签名，在使用这个数据之前需要检查签名。这里给出的扩展例子可以在文件 `hmac_pickle.py` 中找到。

第一步是建立一个函数以计算一个串的摘要，另外实例化一个简单的类，并通过一个通信通道传递。

```
import hashlib
import hmac
try:
    import cPickle as pickle
except:
    import pickle
import pprint
from StringIO import StringIO

def make_digest(message):
    "Return a digest for the message."
    hash = hmac.new('secret-shared-key-goes-here',
                    message,
                    hashlib.sha1)
```



```
return hash.hexdigest()
```

```
class SimpleObject(object):
    """A very simple class to demonstrate checking digests before
    unpickling.
    """
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name
```

接下来，创建一个 StringIO 缓冲区来表示这个套接字或管道。这个例子对数据流使用了一种简单但易于解析的格式。首先写出摘要以及数据长度，后面是一个换行符。接下来是对象的串行化表示（由 pickle 生成）。实际的系统可能不希望依赖于一个长度值，因为如果摘要不正确，这个长度可能也是错误的。可能更合适的做法是使用真实数据中不太可能出现的某个终止符序列。

然后这个示例程序向流写两个对象。写第一个对象时使用正确的摘要值。

```
# Simulate a writable socket or pipe with StringIO
out_s = StringIO()

# Write a valid object to the stream:
# digest\nlength\npickle
o = SimpleObject('digest matches')
pickled_data = pickle.dumps(o)
digest = make_digest(pickled_data)
header = '%s %s' % (digest, len(pickled_data))
print 'WRITING:', header
out_s.write(header + '\n')
out_s.write(pickled_data)
```

再用一个不正确的摘要将第二个对象写至流，这是为另外某个数据计算的摘要，而不是由 pickle 生成。

```
# Write an invalid object to the stream
o = SimpleObject('digest does not match')
pickled_data = pickle.dumps(o)
digest = make_digest('not the pickled data at all')
header = '%s %s' % (digest, len(pickled_data))
print '\nWRITING:', header
out_s.write(header + '\n')
out_s.write(pickled_data)

out_s.flush()
```

既然数据在 StringIO 缓冲区中，可以将它再次读出。首先读取包含摘要和数据长度的数

据行。然后使用得到的长度值读取其余数据。`pickle.load()` 可以直接从流读数据，不过这有一个假设，认为它是一个可信的数据流，而这个数据还不能保证足够可信来解除 pickle。可以将 pickle 作为一个串从流读取，而不是具体将对象解除 pickle，这样会更为安全。

```
# Simulate a readable socket or pipe with StringIO
in_s = StringIO(out_s.getvalue())

# Read the data
while True:
    first_line = in_s.readline()
    if not first_line:
        break
    incoming_digest, incoming_length = first_line.split(' ')
    incoming_length = int(incoming_length)
    print '\nREAD:', incoming_digest, incoming_length

    incoming_pickled_data = in_s.read(incoming_length)
```

pickle 数据一旦在内存中，可以重新计算摘要值，并与所读取的数据比较。如果摘要匹配，就可以信任这个数据，对其解除 pickle。

```
actual_digest = make_digest(incoming_pickled_data)
print 'ACTUAL:', actual_digest

if incoming_digest != actual_digest:
    print 'WARNING: Data corruption'
else:
    obj = pickle.loads(incoming_pickled_data)
    print 'OK:', obj
```

输出显示第一个对象通过验证，另外不出所料，认为第二个对象“已被破坏”。

```
$ python hmac_pickle.py
```

```
WRITING: 387632cfa3d18cd19bdfe72b61ac395dfcdc87c9 124
```

```
WRITING: b01b209e28d7e053408ebe23b90fe5c33bc6a0ec 131
```

```
READ: 387632cfa3d18cd19bdfe72b61ac395dfcdc87c9 124
```

```
ACTUAL: 387632cfa3d18cd19bdfe72b61ac395dfcdc87c9
```

```
OK: digest matches
```

```
READ: b01b209e28d7e053408ebe23b90fe5c33bc6a0ec 131
```

```
ACTUAL: dec53calad3f4b657dd81d514f17f735628b6828
```

```
WARNING: Data corruption
```

参见：

`hmac` (<http://docs.python.org/library/hmac.html>) 这个模块的标准库文档。

RFC 2104 (<http://tools.ietf.org/html/rfc2104.html>) HMAC: 基于密钥的散列来完成消息认证。

hashlib (9.1 节) hashlib 模块提供了 MD5 和 SHA1 散列生成器。

pickle (7.1 节) 串行化库。

WikiPedia: MD5 (<http://en.wikipedia.org/wiki/MD5>) MD5 散列算法的描述。

Authenticating to Amazon S3 Web Service(http://docs.amazonwebservices.com/AmazonS3/2006-03-01/index.html?S3_Authentication.html) 使用 HMAC-SHA1 签名的凭证对 S3 完成认证的有关说明。



第 10 章

进程与线程

Python 提供了一些复杂的工具，用于管理使用进程和线程的并发操作。通过应用这些技术，使用这些模块并发地运行作业的不同部分，即使是一些相当简单的程序，也可以更快地运行。

`subprocess` 提供了一个 API，可以创建子进程并与之通信。如果一个程序需要生产或利用文本，这个模块尤其有帮助，因为这个 API 支持通过新进程的标准输入和输出通道来回传递数据。

`signal` 模块提供了 UNIX 信号机制，可以向其他进程发送事件。信号会异步处理，通常信号到来时要中断程序正在做的工作。作为一个粗粒度的消息系统，信号很有用，不过其他进程内通信技术则更为可靠，而且可以传递更复杂的消息。

`threading` 包含一个面向对象的高层 API 来处理 Python 的并发性。`Thread` 对象在同一个进程中并发地运行，并共享内存。对于 I/O 受限而不是 CPU 受限的任务来说，使用线程是实现这种任务缩放的一种简单方法。`multiprocessing` 模块是 `threading` 的镜像，只是并不是提供 `Thread` 类，而是提供一个 `Process`。每个 `Process` 是无共享内存的真正的系统进程，不过 `multiprocessing` 提供了一些特性，可以共享数据并在进程间传递消息。很多情况下，从线程转换为进程很简单，只需修改几个 `import` 语句。

10.1 subprocess——创建附加进程

作用：创建附加进程，并与之通信。

Python 版本：2.4 及以后版本

`subprocess` 模块提供了一种一致的方法来创建和处理附加进程。与标准库中的其他模块相比，它提供了一个更高级的接口，用以替换 `os.system()`、`os.spawnv()`、`os` 和 `popen2` 模块中的 `popen()` 函数，以及 `commands()`。为了更易于比较 `subprocess` 和其他模块，本节中的很多例子将重新创建 `os` 和 `popen2` 中使用的例子。

`subprocess` 模块定义了一个类 `Popen`，还定义了使用这个类的一些包装器函数。`Popen` 的构造函数根据一些参数建立新进程，使父进程可以通过管道与之通信。相对于它替换的其他模块和函数，`subprocess` 能提供其全部功能，甚至更多。对于所有情况，这个 API 用法都一致，很多需要开销的额外步骤（如关闭额外的文件描述符，以及确保管道关闭）都已“内置”，而不需要由应用代码单独处理。

注意：UNIX 和 Windows 上使用的 API 大致相同，不过底层实现稍有不同。这里显示的所有例子都已经在 Mac OS X 上测试。非 UNIX OS 上的行为可能会有不同。

10.1.1 运行外部命令

要运行一个外部命令，但不采用 `os.system()` 的方式与之交互，可以使用 `call()` 函数。

```
import subprocess
```

```
# Simple command  
subprocess.call(['ls', '-l'])
```

命令行参数作为一个字符串列表传入，这样就无须对引号或其他可能由 shell 解释的特殊字符转义。

```
$ python subprocess_os_system.py
```

```
__init__.py  
index.rst  
interaction.py  
repeater.py  
signal_child.py  
signal_parent.py  
subprocess_check_call.py  
subprocess_check_output.py  
subprocess_check_output_error.py  
subprocess_check_output_error_trap_output.py  
subprocess_os_system.py  
subprocess_pipes.py  
subprocess_popen2.py  
subprocess_popen3.py  
subprocess_popen4.py  
subprocess_popen_read.py  
subprocess_popen_write.py  
subprocess_shell_variables.py  
subprocess_signal_parent_shell.py  
subprocess_signal_setsid.py
```

将 `shell` 参数设置为 `true` 值会使 `subprocess` 创建一个中间 shell 进程，由这个进程运行命令。默认情况下会直接运行命令。

```
import subprocess
```

```
# Command with shell expansion  
subprocess.call('echo $HOME', shell=True)
```

使用一个中间 shell 意味着在运行命令之前会先处理命令串中的变量、glob 模式以及其他特殊 shell 特性。

```
$ python subprocess_shell_variables.py
```

```
/Users/dhellmann
```

错误处理

call() 的返回值是程序的退出码。调用者要负责解释这个返回值来检测错误。check_call() 函数的工作类似于 call(), 只不过除了检查退出码外, 如果指示发生了一个错误, 则会产生一个 CalledProcessError 异常。

```
import subprocess

try:
    subprocess.check_call(['false'])
except subprocess.CalledProcessError as err:
    print 'ERROR:', err
```

false 命令退出时总有一个非 0 的状态码, check_call() 会把它解释为一个错误。

```
$ python subprocess_check_call.py
```

```
ERROR: Command '['false']' returned nonzero exit status 1
```

捕获输出

对于 call() 启动的进程, 其标准输入和输出通道会绑定到父进程的输入和输出。这说明调用程序无法捕获命令的输出。可以使用 check_output() 捕获输出, 以备以后处理。

```
import subprocess

output = subprocess.check_output(['ls', '-l'])
print 'Have %d bytes in output' % len(output)
print output
```

ls -l 命令会成功运行, 所以它打印到标准输出的文本会被捕获并返回。

```
$ python subprocess_check_output.py
```

```
Have 462 bytes in output
__init__.py
index.rst
interaction.py
repeater.py
signal_child.py
signal_parent.py
subprocess_check_call.py
subprocess_check_output.py
subprocess_check_output_error.py
subprocess_check_output_error_trap_output.py
subprocess_os_system.py
subprocess_pipes.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
```



```

subprocess_popen_read.py
subprocess_popen_write.py
subprocess_shell_variables.py
subprocess_signal_parent_shell.py
subprocess_signal_setsid.py

```

下一个例子在一个子 shell 中运行一系列命令。在命令返回一个错误码并退出之前，消息会发送到标准输出和标准错误输出。

```

import subprocess

try:
    output = subprocess.check_output(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        shell=True,
    )
except subprocess.CalledProcessError as err:
    print 'ERROR:', err
else:
    print 'Have %d bytes in output' % len(output)
    print output

```

发送到标准错误输出的消息会打印到控制台，不过发送到标准输出的消息会隐藏。

```
$ python subprocess_check_output_error.py
```

```

to stderr
ERROR: Command 'echo to stdout; echo to stderr 1>&2; exit 1' returned
nonzero exit status 1

```

为了避免通过 `check_output()` 运行的命令将错误消息写至控制台，可以设置 `stderr` 参数为常量 `STDOUT`。

```

import subprocess

try:
    output = subprocess.check_output(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        shell=True,
        stderr=subprocess.STDOUT,
    )
except subprocess.CalledProcessError as err:
    print 'ERROR:', err
else:
    print 'Have %d bytes in output' % len(output)
    print output

```

现在错误和标准输出通道合并在一起，所以如果命令打印错误消息，它们会被捕获，而不会发送至控制台。

```
$ python subprocess_check_output_error_trap_output.py
```

```
ERROR: Command 'echo to stdout; echo to stderr 1>&2; exit 1' returned
nonzero exit status 1
```

10.1.2 直接处理管道

函数 `call()`、`check_call()` 和 `check_output()` 都是 `Popen` 类的包装器。直接使用 `Popen` 会对如何运行命令以及如何处理其输入和输出流有更多控制。例如，通过为 `stdin`、`stdout` 和 `stderr` 传递不同的参数，可以模仿 `os.popen()` 的不同变种。

与进程的单向通信

要运行一个进程并读取它的所有输出，可以设置 `stdout` 值为 `PIPE` 并调用 `communicate()`。

```
import subprocess
```

```
print 'read:'
proc = subprocess.Popen(['echo', '"to stdout"'],
                        stdout=subprocess.PIPE,
                        )
stdout_value = proc.communicate()[0]
print '\tstdout:', repr(stdout_value)
```

这与 `popen()` 的工作类似，只不过 `Popen` 实例会在内部管理数据读取。

```
$ python subprocess_popen_read.py
```

```
read:
    stdout: '"to stdout"\n'
```

要建立一个管道，以便调用程序写数据，可以设置 `stdin` 为 `PIPE`。

```
import subprocess
```

```
print 'write:'
proc = subprocess.Popen(['cat', '-'],
                        stdin=subprocess.PIPE,
                        )
proc.communicate('\tstdin: to stdin\n')
```

要将数据一次性发送到进程的标准输入通道，可以把数据传递到 `communicate()`。这与基于模式 `'w'` 使用 `popen()` 类似。

```
$ python -u subprocess_popen_write.py
```

```
write:
    stdin: to stdin
```

与进程的双向通信

要建立 `Popen` 实例同时完成读写，可以结合使用前面几项技术。

```
import subprocess
```

```

print 'popen2:'

proc = subprocess.Popen(['cat', '-'],
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        )
msg = 'through stdin to stdout'
stdout_value = proc.communicate(msg)[0]
print '\tpass through:', repr(stdout_value)

```

这会建立管道，类似于 `popen2()`。

```
$ python -u subprocess_popen2.py
```

```

popen2:
    pass through: 'through stdin to stdout'

```

捕获错误输出

还可以监视 `stdout` 和 `stderr` 数据流，类似于 `popen3()`。

```

import subprocess

print 'popen3:'
proc = subprocess.Popen('cat -; echo "to stderr" 1>&2',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        )
msg = 'through stdin to stdout'
stdout_value, stderr_value = proc.communicate(msg)
print '\tpass through:', repr(stdout_value)
print '\tstderr      :', repr(stderr_value)

```

从 `stderr` 读取数据与读取 `stdout` 是一样的。传入 `PIPE` 则告诉 `Popen` 关联到通道，`communicate()` 在返回之前会从这个通道读取所有数据。

```
$ python -u subprocess_popen3.py
```

```

popen3:
    pass through: 'through stdin to stdout'
    stderr      : 'to stderr\n'

```

结合常规和错误输出

要把错误输出从进程定向到标准输出通道，`stderr` 要使用 `STDOUT` 而不是 `PIPE`。

```

import subprocess

print 'popen4:'

```



```

proc = subprocess.Popen('cat -; echo "to stderr" 1>&2',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.STDOUT,
                        )
msg = 'through stdin to stdout\n'
stdout_value, stderr_value = proc.communicate(msg)
print '\tcombined output:', repr(stdout_value)
print '\tstderr value   : ', repr(stderr_value)

```

以这种方式结合输出类似于 `popen4()`。

```
$ python -u subprocess_popen4.py
```

```

popen4:
    combined output: 'through stdin to stdout\nto stderr\n'
    stderr value   : None

```

10.1.3 连接管道段

多个命令可以连接为一个管线 (pipeline)，这类似于 UNIX shell 的做法，即创建单独的 Popen 实例，把它们的输入和输出串链在一起。一个 Popen 实例的 `stdout` 属性用作管线中下一个 Popen 实例的 `stdin` 参数，而不是常量 PIPE。输出从管线中最后一个命令的 `stdout` 句柄读取。

```

import subprocess

cat = subprocess.Popen(['cat', 'index.rst'],
                      stdout=subprocess.PIPE,
                      )

grep = subprocess.Popen(['grep', '.. include::'],
                      stdin=cat.stdout,
                      stdout=subprocess.PIPE,
                      )

cut = subprocess.Popen(['cut', '-f', '3', '-d:'],
                      stdin=grep.stdout,
                      stdout=subprocess.PIPE,
                      )

end_of_pipe = cut.stdout
print 'Included files:'
for line in end_of_pipe:
    print '\t', line.strip()

```

这个例子重新生成以下命令行。

```
cat index.rst | grep ".. include" | cut -f 3 -d:
```



这个管线读取本节的 reStructuredText 源文件，查找所有包含其他文件的文本行。然后打印出所包含的这些文件的文件名。

```
$ python -u subprocess_pipes.py
```

Included files:

```
subprocess_os_system.py
subprocess_shell_variables.py
subprocess_check_call.py
subprocess_check_output.py
subprocess_check_output_error.py
subprocess_check_output_error_trap_output.py
subprocess_popen_read.py
subprocess_popen_write.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
subprocess_pipes.py
repeater.py
interaction.py
signal_child.py
signal_parent.py
subprocess_signal_parent_shell.py
subprocess_signal_setsid.py
```

10.1.4 与其他命令交互

前面的所有例子都假设交互量是有限的。communicate() 方法读取所有输出，返回之前要等待子进程退出。也可以在程序运行时从 Popen 实例使用的单个管道句柄增量地进行读写。这个技术可以用一个简单的应答程序来说明，这个程序从标准输入读，并写至标准输出。

下一个例子中使用脚本 repeater.py 作为子进程。它从 stdin 读取，将值写至 stdout，一次处理一行，直到再没有更多输入。开始和停止时它还会向 stderr 写一个消息，显示子进程的生命期。

```
import sys

sys.stderr.write('repeater.py: starting\n')
sys.stderr.flush()

while True:
    next_line = sys.stdin.readline()
    if not next_line:
        break
    sys.stdout.write(next_line)
    sys.stdout.flush()

sys.stderr.write('repeater.py: exiting\n')
sys.stderr.flush()
```

下一个交互例子将采用不同方式使用 Popen 实例的 stdin 和 stdout 文件句柄。在第一个例子中，将把一组 5 个数写至进程的 stdin，每写一个数就读回下一行输出。第二个例子中仍然写同样的 5 个数，但要使用 communicate() 一次读取全部输出。

```
import subprocess

print 'One line at a time:'
proc = subprocess.Popen('python repeater.py',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        )

for i in range(5):
    proc.stdin.write('%d\n' % i)
    output = proc.stdout.readline()
    print output.rstrip()
remainder = proc.communicate()[0]
print remainder
print
print 'All output at once:'
proc = subprocess.Popen('python repeater.py',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        )

for i in range(5):
    proc.stdin.write('%d\n' % i)

output = proc.communicate()[0]
print output
```

对于这两种不同的循环，repeater.py: exiting 行出现在输出的不同位置上。

```
$ python -u interaction.py
```

```
One line at a time:
repeater.py: starting
0
1
2
3
4
repeater.py: exiting
```

```
All output at once:
repeater.py: starting
repeater.py: exiting
```



```
0
1
2
3
4
```

10.1.5 进程间传递信号

os 模块的进程管理例子演示了如何使用 os.fork() 和 os.kill() 在进程间传递信号。由于每个 Popen 实例提供了一个 pid 属性，其中包含子进程的进程 id，所以可以完成类似于 subprocess 的工作。下一个例子结合了两个脚本。这个子进程为 USR1 信号建立了一个信号处理程序。

```
import os
import signal
import time
import sys

pid = os.getpid()
received = False

def signal_usr1(signum, frame):
    "Callback invoked when a signal is received"
    global received
    received = True
    print 'CHILD %6s: Received USR1' % pid
    sys.stdout.flush()

print 'CHILD %6s: Setting up signal handler' % pid
sys.stdout.flush()
signal.signal(signal.SIGUSR1, signal_usr1)
print 'CHILD %6s: Pausing to wait for signal' % pid
sys.stdout.flush()
time.sleep(3)

if not received:
    print 'CHILD %6s: Never received signal' % pid
```

这个脚本作为父进程运行。它启动 signal_child.py，然后发送 USR1 信号。

```
import os
import signal
import subprocess
import time
import sys

proc = subprocess.Popen(['python', 'signal_child.py'])
print 'PARENT      : Pausing before sending signal...'
sys.stdout.flush()
```

```

time.sleep(1)
print 'PARENT      : Signaling child'
sys.stdout.flush()
os.kill(proc.pid, signal.SIGUSR1)

```

输出如下:

```
$ python signal_parent.py
```

```

PARENT      : Pausing before sending signal...
CHILD 11298: Setting up signal handler
CHILD 11298: Pausing to wait for signal
PARENT      : Signaling child
CHILD 11298: Received USR1

```

进程组 / 会话

如果 Popen 创建的进程创建了子进程, 这些子进程不会接收发送给父进程的信号。这说明, 使用 Popen 的 shell 参数时, 很难通过发送 SIGINT 或 SIGTERM 来终止在 shell 中启动的命令。

```

import os
import signal
import subprocess
import tempfile
import time
import sys

script = '''#!/bin/sh
echo "Shell script in process $$"
set -x
python signal_child.py
'''

script_file = tempfile.NamedTemporaryFile('wt')
script_file.write(script)
script_file.flush()

proc = subprocess.Popen(['sh', script_file.name], close_fds=True)
print 'PARENT      : Pausing before signaling %s...' % proc.pid
sys.stdout.flush()
time.sleep(1)
print 'PARENT      : Signaling child %s' % proc.pid
sys.stdout.flush()
os.kill(proc.pid, signal.SIGUSR1)
time.sleep(3)

```

发送信号所用的 pid 与等待信号的 shell 脚本子进程的 pid 不匹配, 因为在这个例子中, 有 3 个不同的进程在交互。

1. 程序 subprocess_signal_parent_shell.py
2. shell 进程, 它在运行主 Python 程序创建的脚本

3. 程序 signal_child.py

```
$ python subprocess_signal_parent_shell.py
```

```
PARENT      : Pausing before signaling 11301...
Shell script in process 11301
+ python signal_child.py
CHILD 11302: Setting up signal handler
CHILD 11302: Pausing to wait for signal
PARENT      : Signaling child 11301
CHILD 11302: Never received signal
```

如果向子进程发送信号但不知道其进程 id，可以使用一个进程组 (process group) 关联这些子进程，使它们能一起收到信号。进程组用 `os.setsid()` 创建，将“会话 id”设置为当前进程的进程 id。所有子进程都会从其父进程继承会话 id，由于只能在由 Popen 及其子进程创建的 shell 中设置，所以不能在创建 Popen 的同一进程中调用 `os.setsid()`。实际上，这个函数要作为 `preexec_fn` 参数传至 Popen，从而在 `fork()` 之后在新进程中运行这个函数，之后才能使用 `exec()` 运行 shell。要向整个进程组发送信号，可以提供 Popen 实例的 `pid` 值来使用 `os.killpg()`。

```
import os
import signal
import subprocess
import tempfile
import time
import sys

script = '''#!/bin/sh
echo "Shell script in process $$"
set -x
python signal_child.py
'''

script_file = tempfile.NamedTemporaryFile('wt')
script_file.write(script)
script_file.flush()

def show_setting_sid():
    print 'Calling os.setsid() from %s' % os.getpid()
    sys.stdout.flush()
    os.setsid()

proc = subprocess.Popen(['sh', script_file.name],
                        close_fds=True,
                        preexec_fn=show_setting_sid,
                        )

print 'PARENT      : Pausing before signaling %s...' % proc.pid
sys.stdout.flush()
```

```

time.sleep(1)
print 'PARENT      : Signaling process group %s' % proc.pid
sys.stdout.flush()
os.killpg(proc.pid, signal.SIGUSR1)
time.sleep(3)

```

事件序列如下：

1. 父程序实例化 Popen。
2. Popen 实例创建一个新进程。
3. 这个新进程运行 os.setsid()。
4. 这个新进程运行 exec() 启动 shell。
5. shell 运行 shell 脚本。
6. shell 脚本再次创建新进程，该进程执行启动 Python。
7. Python 运行 signal_child.py。
8. 父程序使用 shell 的 pid 向进程组传送信号。
9. shell 和 Python 进程接收到信号。
10. shell 忽略这个信号。
11. 运行 signal_child.py 的 Python 进程调用信号处理程序。

```
$ python subprocess_signal_setsid.py
```

```

Calling os.setsid() from 11305
PARENT      : Pausing before signaling 11305...
Shell script in process 11305
+ python signal_child.py
CHILD 11306: Setting up signal handler
CHILD 11306: Pausing to wait for signal
PARENT      : Signaling process group 11305
CHILD 11306: Received USR1

```

参见：

subprocess (<http://docs.python.org/lib/module-subprocess.html>) 这个模块的标准库文档。

UNIX Signals and Process Groups(www.frostbytes.com/~jimf/papers/signals/signals.html) 对 UNIX 信号机制以及进程组如何工作做了很好的描述。

os (17.3 节) 尽管 subprocess 替代了 os 模块中很多处理进程的函数，但现有代码中仍在广泛使用 os 模块中的函数。

signal (10.2 节) 关于使用 signal 模块的更多详细信息。

Advanced Programming in the UNIX(R) Environment(www.amazon.com/Programming-Environment-Addison-Wesley-Professional-Computing/dp/0201433079/ref=pd_bbs_3/002-2842372-4768037?ie=UTF8&s=books&qid=1182098757&sr=8-3) 涵盖了如何处理多个进程的内容，如处理信号、关闭重复的文件描述符，等等。

pipes 标准库中的 UNIX shell 命令管线模板。

10.2 signal——异步系统事件

作用：发送和接收异步系统事件。

Python 版本：1.4 及以后版本

信号是一个操作系统特性，它提供了一个途径可以通知程序发生了一个事件并异步处理这个事件。信号可以由系统本身生成，也可以从一个进程发送到另一个进程。由于信号会中断程序的正常控制流，如果在中间接收到信号，有些操作（特别是 I/O 操作）可能会产生错误。

信号由整数标识，在操作系统 C 首部中定义。Python 在 signal 模块中作为符号提供了适合不同平台的多种信号。本节中的例子使用 SIGINT 和 SIGUSR1。这两个信号一般是为所有 UNIX 和类 UNIX 系统定义的。

注意：使用 UNIX 信号处理程序进行编程并不容易。这里只是一个介绍，并没有涵盖在每一个平台上成功使用信号所需的所有细节。不同版本的 UNIX 有一定程度的标准化，但也确实有一些不同。如果遇到麻烦，可以参考操作系统文档。

10.2.1 接收信号

与其他形式基于事件的编程一样，要通过建立一个回调函数来接收信号，这个回调函数称为信号处理程序（signal handler），它会在出现信号时调用。信号处理程序的参数包括信号编号以及程序被信号中断那一时刻的栈帧。

```
import signal
import os
import time

def receive_signal(signum, stack):
    print 'Received:', signum

# Register signal handlers
signal.signal(signal.SIGUSR1, receive_signal)
signal.signal(signal.SIGUSR2, receive_signal)

# Print the process ID so it can be used with 'kill'
# to send this program signals.
print 'My PID is:', os.getpid()

while True:
    print 'Waiting...'
    time.sleep(3)
```

这个示例脚本会无限循环，每次暂停几秒时间。一个信号到来时，sleep() 调用被中断，信

号处理程序 `receive_signal()` 打印信号编号。信号处理程序返回时，循环继续。

可以使用 `os.kill()` 或 UNIX 命令程序 `kill` 向正在运行的程序发送信号。

```
$ python signal_signal.py

My PID is: 71387
Waiting...
Waiting...
Waiting...
Received: 30
Waiting...
Waiting...
Received: 31
Waiting...
Waiting...
Traceback (most recent call last):
  File "signal_signal.py", line 25, in <module>
    time.sleep(3)
KeyboardInterrupt
```

在一个窗口中运行 `signal_signal.py`，然后在另一个窗口中运行以下命令可以生成前面的输出。

```
$ kill -USR1 $pid
$ kill -USR2 $pid
$ kill -INT $pid
```

10.2.2 获取注册的处理程序

要查看为一个信号注册了哪些信号处理程序，可以使用 `getsignal()`。要将信号编号作为参数传入。返回值是已注册的处理程序，或者是以下某个特殊值：`SIG_IGN`（如果信号被忽略）、`SIG_DFL`（如果使用默认行为）或 `None`（如果从 C 而不是从 Python 注册现有信号处理程序）。

```
import signal

def alarm_received(n, stack):
    return

signal.signal(signal.SIGALRM, alarm_received)

signals_to_names = dict(
    (getattr(signal, n), n)
    for n in dir(signal)
    if n.startswith('SIG') and '_' not in n
)

for s, name in sorted(signals_to_names.items()):
    handler = signal.getsignal(s)
    if handler is signal.SIG_DFL:
```



```

        handler = 'SIG_DFL'
    elif handler is signal.SIG_IGN:
        handler = 'SIG_IGN'
    print '%-10s (%2d):' % (name, s), handler

```

同样地，由于每个操作系统可能定义了不同的信号，其他系统上的输出可能有所不同。以下是 OS X 的输出：

```

$ python signal_getsignal.py

SIGHUP      ( 1): SIG_DFL
SIGINT      ( 2): <built-in function default_int_handler>
SIGQUIT     ( 3): SIG_DFL
SIGILL      ( 4): SIG_DFL
SIGTRAP     ( 5): SIG_DFL
SIGIOT      ( 6): SIG_DFL
SIGEMT      ( 7): SIG_DFL
SIGFPE      ( 8): SIG_DFL
SIGKILL     ( 9): None
SIGBUS      (10): SIG_DFL
SIGSEGV     (11): SIG_DFL
SIGSYS      (12): SIG_DFL
SIGPIPE     (13): SIG_IGN
SIGALRM     (14): <function alarm_received at 0x10045b398>
SIGTERM     (15): SIG_DFL
SIGURG      (16): SIG_DFL
SIGSTOP     (17): None
SIGTSTP     (18): SIG_DFL
SIGCONT     (19): SIG_DFL
SIGCHLD     (20): SIG_DFL
SIGTTIN     (21): SIG_DFL
SIGTTOU     (22): SIG_DFL
SIGIO       (23): SIG_DFL
SIGXCPU     (24): SIG_DFL
SIGXFSZ     (25): SIG_IGN
SIGVTALRM   (26): SIG_DFL
SIGPROF     (27): SIG_DFL
SIGWINCH    (28): SIG_DFL
SIGINFO     (29): SIG_DFL
SIGUSR1     (30): SIG_DFL
SIGUSR2     (31): SIG_DFL

```

10.2.3 发送信号

在 Python 中发送信号的函数是 `os.kill()`。其用法在有关 `os` 模块的 17.3.11 节中介绍。

10.2.4 闹铃

闹铃 (Alarm) 是一种特殊的信号，程序要求操作系统在过去一段时间之后再发出这个信号通知。os 的标准模块文档指出，这对于避免一个 I/O 操作或其他系统调用无限阻塞很有用。

```
import signal
import time

def receive_alarm(signum, stack):
    print 'Alarm :', time.ctime()

# Call receive_alarm in 2 seconds
signal.signal(signal.SIGALRM, receive_alarm)
signal.alarm(2)

print 'Before:', time.ctime()
time.sleep(4)
print 'After :', time.ctime()
```

在这个例子中，sleep() 调用不会完整地持续 4 秒。

```
$ python signal_alarm.py
```

```
Before: Sun Aug 17 10:51:09 2008
Alarm : Sun Aug 17 10:51:11 2008
After : Sun Aug 17 10:51:11 2008
```

10.2.5 忽略信号

要忽略一个信号，需要注册 SIG_IGN 作为处理程序。下面这个脚本将 SIGINT 的默认处理程序替换为 SIG_IGN，并为 SIGUSR1 注册一个处理程序。然后使用 signal.pause() 等待接收一个信号。

```
import signal
import os
import time

def do_exit(sig, stack):
    raise SystemExit('Exiting')

signal.signal(signal.SIGINT, signal.SIG_IGN)
signal.signal(signal.SIGUSR1, do_exit)

print 'My PID:', os.getpid()

signal.pause()
```

正常情况下，SIGINT（用户按下 Ctrl-C 时 shell 会向程序发送这个信号）会产生一个

KeyboardInterrupt。这个例子将忽略 SIGINT，并在发现 SIGUSR1 时产生一个 SystemExit。输出中的每个 ^C 表示每一次尝试使用 Ctrl-C 从终端结束脚本。从另一个终端使用 kill -USR1 72598 才最终退出脚本。

```
$ python signal_ignore.py
```

```
My PID: 72598
^C^C^C^CExiting
```

10.2.6 信号和线程

信号和线程通常不能很好地结合，因为只有进程的主线程可以接收信号。下面的例子建立了一个信号处理程序，它在一个线程中等待信号，而从另一个线程发送信号。

```
import signal
import threading
import os
import time

def signal_handler(num, stack):
    print 'Received signal %d in %s' % \
        (num, threading.currentThread().name)

signal.signal(signal.SIGUSR1, signal_handler)

def wait_for_signal():
    print 'Waiting for signal in', threading.currentThread().name
    signal.pause()
    print 'Done waiting'

# Start a thread that will not receive the signal
receiver = threading.Thread(target=wait_for_signal, name='receiver')
receiver.start()
time.sleep(0.1)

def send_signal():
    print 'Sending signal in', threading.currentThread().name
    os.kill(os.getpid(), signal.SIGUSR1)

sender = threading.Thread(target=send_signal, name='sender')
sender.start()
sender.join()

# Wait for the thread to see the signal (not going to happen!)
print 'Waiting for', receiver.name
signal.alarm(2)
receiver.join()
```

信号处理程序都在主线程中注册，因为这是 signal 模块 Python 实现的一个要求，不论底层平台对于结合线程和信号提供怎样的支持都有这个要求。尽管接收者线程调用了 signal.pause()，但它不会接收信号。这个例子接近结束时的 signal.alarm(2) 调用避免了无限阻塞，因为接收者线程永远不会退出。

```
$ python signal_threads.py
```

```
Waiting for signal in receiver
Sending signal in sender
Received signal 30 in MainThread
Waiting for receiver
Alarm clock
```

尽管在任何线程中都能设置闹铃，但总是由主线程接收。

```
import signal
import time
import threading

def signal_handler(num, stack):
    print time.ctime(), 'Alarm in', threading.currentThread().name

signal.signal(signal.SIGALRM, signal_handler)

def use_alarm():
    t_name = threading.currentThread().name
    print time.ctime(), 'Setting alarm in', t_name
    signal.alarm(1)
    print time.ctime(), 'Sleeping in', t_name
    time.sleep(3)
    print time.ctime(), 'Done with sleep in', t_name

# Start a thread that will not receive the signal
alarm_thread = threading.Thread(target=use_alarm,
                                name='alarm_thread')
alarm_thread.start()
time.sleep(0.1)

# Wait for the thread to see the signal (not going to happen!)
print time.ctime(), 'Waiting for', alarm_thread.name
alarm_thread.join()

print time.ctime(), 'Exiting normally'
```

闹铃不会中止 use_alarm() 中的 sleep() 调用。

```
$ python signal_threads_alarm.py
```

```
Sun Nov 28 14:26:51 2010 Setting alarm in alarm_thread
```

```
Sun Nov 28 14:26:51 2010 Sleeping in alarm_thread
Sun Nov 28 14:26:52 2010 Waiting for alarm_thread
Sun Nov 28 14:26:54 2010 Done with sleep in alarm_thread
Sun Nov 28 14:26:54 2010 Alarm in MainThread
Sun Nov 28 14:26:54 2010 Exiting normally
```

参见:

signal (<http://docs.python.org/lib/module-signal.html>) 这个模块的标准库文档。

17.3.11 节 kill() 函数可以用来在进程之间发送信号。

10.3 threading——管理并发操作

作用: 建立在 thread 模块之上, 可以更容易地管理多个执行线程。

Python 版本: 1.5.2 及以后版本

通过使用线程, 程序可以在同一个进程空间并发地运行多个操作。threading 模块建立在 thread 的底层特性基础之上, 可以更容易地完成线程处理。

10.3.1 Thread 对象

要使用 Thread, 最简单的方法就是用目标函数实例化一个 Thread 对象, 并调用 start() 让它开始工作。

```
import threading

def worker():
    """thread worker function"""
    print 'Worker'
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

输出有 5 行, 每一行都是 “Worker”:

```
$ python threading_simple.py
```

```
Worker
Worker
Worker
Worker
Worker
```

如果能够创建一个线程, 并向它传递参数告诉它要完成什么工作, 这会很有用。任何类型的对象都可以作为参数传递到线程。下面的例子传递了一个数, 线程将打印出这个数。

```

import threading

def worker(num):
    """thread worker function"""
    print 'Worker: %s' % num
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()

```

现在整数参数会包含在各线程打印的消息中:

```
$ python -u threading_simpleargs.py
```

```

Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4

```

10.3.2 确定当前线程

使用参数来标识或命名线程很麻烦,也没有必要。每个 Thread 实例都有一个名称,它有一个默认值,可以在创建线程时改变。如果服务器进程由处理不同操作的多个服务线程构成,在这样的服务器进程中,对线程命名就很有用。

```

import threading
import time

def worker():
    print threading.currentThread().getName(), 'Starting'
    time.sleep(2)
    print threading.currentThread().getName(), 'Exiting'

def my_service():
    print threading.currentThread().getName(), 'Starting'
    time.sleep(3)
    print threading.currentThread().getName(), 'Exiting'

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()

```

调试输出的每一行中包含有当前线程的名称。线程名称列为“Thread-1”的行对应未命名的线程 w2。

```
$ python -u threading_names.py
```

```
worker Starting
Thread-1 Starting
my_service Starting
worker Exiting
Thread-1 Exiting
my_service Exiting
```

大多数程序并不使用 print 来进行调试。logging 模块支持将线程名嵌入到各个日志消息中（使用格式化代码 `%(threadName)s`）。通过将线程名包含在日志消息中，这样就能跟踪这些消息的来源。

```
import logging
import threading
import time

logging.basicConfig(
    level=logging.DEBUG,
    format='[% (levelname)s] (% (threadName)s)-10s) % (message)s',
)

def worker():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

def my_service():
    logging.debug('Starting')
    time.sleep(3)
    logging.debug('Exiting')

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()
```

logging 也是线程安全的，所以来自不同线程的消息在输出中会有所区分。

```
$ python threading_names_log.py
```

```
[DEBUG] (worker      ) Starting
[DEBUG] (Thread-1    ) Starting
```



```
[DEBUG] (my_service) Starting
[DEBUG] (worker    ) Exiting
[DEBUG] (Thread-1  ) Exiting
[DEBUG] (my_service) Exiting
```

10.3.3 守护与非守护线程

到目前为止，示例程序都隐含地等待所有线程完成工作之后才退出。程序有时会创建一个线程作为守护线程（daemon），这个线程可以一直运行而不阻塞主程序退出。如果一个服务无法用一种容易的方法来中断线程，或者希望线程工作到一半时中止而不损失或破坏数据（如为一个服务监控工具生成“心跳”的线程），对于这些服务，使用守护线程就很有用。要标志一个线程为守护线程，需要调用其 `setDaemon()` 方法并提供参数 `True`。默认情况下线程不作为守护线程。

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

输出中没有守护线程的“Exiting”消息，因为在守护线程从其 2 秒的睡眠时间唤醒之前，所有非守护线程（包括主线程）已经退出。

```
$ python threading_daemon.py
```

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
```

要等待一个守护线程完成工作，需要使用 `join()` 方法。

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join()
t.join()
```

使用 `join()` 等待守护线程退出，这意味着它将有机会生成它的“Exiting”消息。

```
$ python threading_daemon_join.py
```

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
(daemon    ) Exiting
```

默认情况下，`join()` 会无限阻塞。还可以传入一个浮点数值，表示等待线程变为不活动所需的时间（秒数）。即使线程在这个时间段内未完成，`join()` 也会返回。

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )
```

```

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join(1)
print 'd.isAlive()', d.isAlive()
t.join()

```

由于传入的超时时间小于守护线程睡眠的时间，所以 join() 返回之后这个线程仍“存活”。

```
$ python threading_daemon_join_timeout.py
```

```

(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
d.isAlive() True

```

10.3.4 列举所有线程

没有必要为所有守护线程维护一个显式句柄来确保它们在退出主进程之前已经完成。enumerate() 会返回活动 Thread 实例的一个列表。这个列表也包括当前线程，由于等待当前线程结束会引入一种死锁情况，所以必须将其跳过。

```

import random
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

def worker():
    """thread worker function"""
    t = threading.currentThread()

```

```

        pause = random.randint(1,5)
        logging.debug('sleeping %s', pause)
        time.sleep(pause)
        logging.debug('ending')
        return

    for i in range(3):
        t = threading.Thread(target=worker)
        t.setDaemon(True)
        t.start()

    main_thread = threading.currentThread()
    for t in threading.enumerate():
        if t is main_thread:
            continue
        logging.debug('joining %s', t.getName())
        t.join()

```

由于工作线程睡眠的时间随机，所以这个程序的输出可能有变化。

```
$ python threading_enumerate.py
```

```

(Thread-1 ) sleeping 5
(Thread-2 ) sleeping 4
(Thread-3 ) sleeping 2
(MainThread) joining Thread-1
(Thread-3 ) ending
(Thread-2 ) ending
(Thread-1 ) ending
(MainThread) joining Thread-2
(MainThread) joining Thread-3

```

10.3.5 派生线程

开始时，Thread 要完成一些基本初始化，然后调用其 run() 方法，这会调用传递到构造函数的目标函数。要创建 Thread 的一个子类，需要覆盖 run() 来完成所需的工作。

```

import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

class MyThread(threading.Thread):

    def run(self):
        logging.debug('running')
        return

```

```
for i in range(5):
    t = MyThread()
    t.start()
```

run() 的返回值将忽略。

```
$ python threading_subclass.py
```

```
(Thread-1 ) running
(Thread-2 ) running
(Thread-3 ) running
(Thread-4 ) running
(Thread-5 ) running
```

由于传递到 Thread 构造函数的 args 和 kwargs 值保存在私有变量中（这些变量名都有前缀 '_'），所以不能很容易地从子类访问这些值。要向一个定制的线程类型传递参数，需要重新定义构造函数，将这些值保存在子类可见的一个实例属性中。

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

class MyThreadWithArgs(threading.Thread):

    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, verbose=None):
        threading.Thread.__init__(self, group=group,
                                   target=target,
                                   name=name,
                                   verbose=verbose)

        self.args = args
        self.kwargs = kwargs
        return

    def run(self):
        logging.debug('running with %s and %s',
                      self.args, self.kwargs)

        return

for i in range(5):
    t = MyThreadWithArgs(args=(i,),
                          kwargs={'a': 'A', 'b': 'B'})
    t.start()
```

MyThreadWithArgs 使用的 API 与 Thread 相同，不过这个类可以轻松地修改构造函数方

法，取更多或与线程用途更直接相关的不同参数，这一点类似于其他定制类。

```
$ python threading_subclass_args.py

(Thread-1 ) running with (0,) and {'a': 'A', 'b': 'B'}
(Thread-2 ) running with (1,) and {'a': 'A', 'b': 'B'}
(Thread-3 ) running with (2,) and {'a': 'A', 'b': 'B'}
(Thread-4 ) running with (3,) and {'a': 'A', 'b': 'B'}
(Thread-5 ) running with (4,) and {'a': 'A', 'b': 'B'}
```

10.3.6 定时器线程

有时出于某种原因需要派生 Thread, Timer 就是这样一个例子, Timer 也包含在 threading 中。Timer 在一个延迟之后开始工作，可以在这个延迟期间内的任意时刻取消。

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

def delayed():
    logging.debug('worker running')
    return

t1 = threading.Timer(3, delayed)
t1.setName('t1')
t2 = threading.Timer(3, delayed)
t2.setName('t2')

logging.debug('starting timers')
t1.start()
t2.start()

logging.debug('waiting before canceling %s', t2.getName())
time.sleep(2)
logging.debug('canceling %s', t2.getName())
t2.cancel()
logging.debug('done')
```

第二个定时器永远不会运行，第一个定时器会在其余的主程序完成之后运行。由于这不是一个守护线程，主线程完成时它会隐式退出。

```
$ python threading_timer.py

(MainThread) starting timers
(MainThread) waiting before canceling t2
```

```
(MainThread) canceling t2
(MainThread) done
(t1         ) worker running
```

10.3.7 线程间传送信号

尽管使用多线程的目的是并发地运行单个操作，不过有时也需要在两个或多个线程中同步操作。事件对象是实现线程间安全通信的一种简单方法。Event 管理一个内部标志，调用者可以用 set() 和 clear() 方法控制这个标志。其他线程可以使用 wait() 暂停，直到设置这个标志，其效果就是在允许继续之前阻塞线程的运行。

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    logging.debug('wait_for_event starting')
    event_is_set = e.wait()
    logging.debug('event set: %s', event_is_set)

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    while not e.isSet():
        logging.debug('wait_for_event_timeout starting')
        event_is_set = e.wait(t)
        logging.debug('event set: %s', event_is_set)
        if event_is_set:
            logging.debug('processing event')
        else:
            logging.debug('doing other work')

e = threading.Event()
t1 = threading.Thread(name='block',
                      target=wait_for_event,
                      args=(e,))

t1.start()

t2 = threading.Thread(name='nonblock',
                      target=wait_for_event_timeout,
                      args=(e, 2))

t2.start()
```

```

logging.debug('Waiting before calling Event.set()')
time.sleep(3)
e.set()
logging.debug('Event is set')

```

`wait()` 方法取一个参数，表示在超时之前等待事件的时间（秒数）。它会返回一个布尔值，指示事件是否已设置，从而使调用者知道 `wait()` 为什么返回。可以对事件单独地使用 `isSet()` 方法而不必担心阻塞。

在这个例子中，`wait_for_event_timeout()` 将检查事件状态而不会无限阻塞。`wait_for_event()` 在 `wait()` 调用处阻塞，事件状态改变之前它不会返回。

```

$ python threading_event.py

(block      ) wait_for_event starting
(nonblock   ) wait_for_event_timeout starting
(MainThread) Waiting before calling Event.set()
(nonblock   ) event set: False
(nonblock   ) doing other work
(nonblock   ) wait_for_event_timeout starting
(MainThread) Event is set
(block      ) event set: True
(nonblock   ) event set: True
(nonblock   ) processing event

```

10.3.8 控制资源访问

除了同步线程操作之外，还有一点很重要，要能够控制对共享资源的访问，从而避免破坏或丢失数据。Python 的内置数据结构（列表、字典等等）是线程安全的，这是 Python 使用原子字节码来管理这些数据结构的一个副作用（更新过程中不会释放 GIL）。Python 中实现的其他数据结构或更简单的类型（如整数和浮点数）则没有这个保护。要保证同时安全地访问一个对象，可以使用一个 `Lock` 对象。

```

import logging
import random
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

class Counter(object):
    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.value = start

```



```

def increment(self):
    logging.debug('Waiting for lock')
    self.lock.acquire()
    try:
        logging.debug('Acquired lock')
        self.value = self.value + 1
    finally:
        self.lock.release()

def worker(c):
    for i in range(2):
        pause = random.random()
        logging.debug('Sleeping %0.02f', pause)
        time.sleep(pause)
        c.increment()
    logging.debug('Done')

counter = Counter()
for i in range(2):
    t = threading.Thread(target=worker, args=(counter,))
    t.start()

logging.debug('Waiting for worker threads')
main_thread = threading.currentThread()
for t in threading.enumerate():
    if t is not main_thread:
        t.join()
logging.debug('Counter: %d', counter.value)

```

在这个例子中，worker() 函数使一个 Counter 实例递增，这个实例管理着一个 Lock，来避免两个线程同时改变其内部状态。如果没有使用 Lock，就有可能丢失一次对 value 属性的修改。

```
$ python threading_lock.py
```

```

(Thread-1 ) Sleeping 0.94
(Thread-2 ) Sleeping 0.32
(MainThread) Waiting for worker threads
(Thread-2 ) Waiting for lock
(Thread-2 ) Acquired lock
(Thread-2 ) Sleeping 0.54
(Thread-1 ) Waiting for lock
(Thread-1 ) Acquired lock
(Thread-1 ) Sleeping 0.84
(Thread-2 ) Waiting for lock
(Thread-2 ) Acquired lock
(Thread-2 ) Done
(Thread-1 ) Waiting for lock
(Thread-1 ) Acquired lock

```



```
(Thread-1 ) Done
(MainThread) Counter: 4
```

要查看是否有另一个线程请求这个锁而不影响当前线程，可以向 `acquire()` 的 `blocking` 参数传入 `False`。在下一个例子中，`worker()` 想要得到 3 次锁，并统计为得到锁所尝试的次数。与此同时，`lock_holder()` 在占有和释放锁之间循环，每个状态会短暂的暂停，以模拟负载情况。

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def lock_holder(lock):
    logging.debug('Starting')
    while True:
        lock.acquire()
        try:
            logging.debug('Holding')
            time.sleep(0.5)
        finally:
            logging.debug('Not holding')
            lock.release()
            time.sleep(0.5)
    return

def worker(lock):
    logging.debug('Starting')
    num_tries = 0
    num_acquires = 0
    while num_acquires < 3:
        time.sleep(0.5)
        logging.debug('Trying to acquire')
        have_it = lock.acquire(0)
        try:
            num_tries += 1
            if have_it:
                logging.debug('Iteration %d: Acquired',
                              num_tries)
                num_acquires += 1
            else:
                logging.debug('Iteration %d: Not acquired',
                              num_tries)
        finally:
            if have_it:
                lock.release()
```

```

logging.debug('Done after %d iterations', num_tries)

lock = threading.Lock()

holder = threading.Thread(target=lock_holder,
                          args=(lock,),
                          name='LockHolder')
holder.setDaemon(True)
holder.start()
worker = threading.Thread(target=worker,
                          args=(lock,),
                          name='Worker')
worker.start()

```

worker() 需要超过 3 次迭代才能得到 3 次锁。

```
$ python threading_lock_noblock.py
```

```

(LockHolder) Starting
(LockHolder) Holding
(Worker    ) Starting
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 1: Acquired
(LockHolder) Holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 2: Not acquired
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 3: Acquired
(LockHolder) Holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 4: Not acquired
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 5: Acquired
(Worker    ) Done after 5 iterations

```

再入锁

正常的 Lock 对象不能请求多次，即使是由同一个线程请求。如果一个锁被同一调用链中的多个函数访问，这个限制可能会引入我们不希望的副作用。

```
import threading
```

```
lock = threading.Lock()
```

```
print 'First try:', lock.acquire()
print 'Second try:', lock.acquire(0)
```

在这里，对第二个 `acquire()` 调用给定超时值为 0，以避免阻塞，因为锁已经被第一个调用获得。

```
$ python threading_lock_reacquire.py
```

```
First try : True
Second try: False
```

如果同一个线程的不同代码需要“重新获得”锁，在这种情况下则要使用 `RLock`。

```
import threading
```

```
lock = threading.RLock()
```

```
print 'First try :', lock.acquire()
print 'Second try:', lock.acquire(0)
```

与前面的例子相比，对代码惟一的修改就是用 `RLock` 替换 `Lock`。

```
$ python threading_rlock.py
```

```
First try : True
Second try: 1
```

锁作为上下文管理器

锁实现了上下文管理器 API，并与 `with` 语句兼容。使用 `with` 则不再需要显式地获得和释放锁。

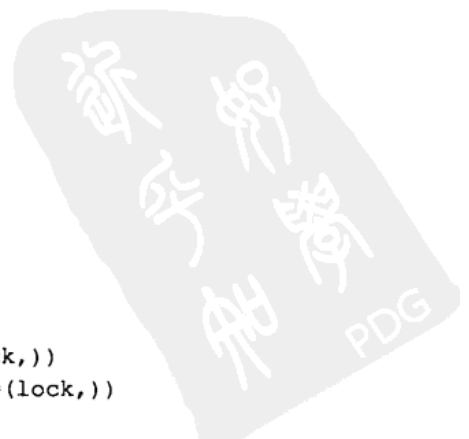
```
import threading
import logging
```

```
logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )
```

```
def worker_with(lock):
    with lock:
        logging.debug('Lock acquired via with')
def worker_no_with(lock):
    lock.acquire()
    try:
        logging.debug('Lock acquired directly')
    finally:
        lock.release()
```

```
lock = threading.Lock()
w = threading.Thread(target=worker_with, args=(lock,))
nw = threading.Thread(target=worker_no_with, args=(lock,))

w.start()
nw.start()
```



函数 `worker_with()` 和 `worker_no_with()` 分别用相应的方式管理锁。

```
$ python threading_lock_with.py
```

```
(Thread-1 ) Lock acquired via with
(Thread-2 ) Lock acquired directly
```

10.3.9 同步线程

除了使用 `Event`，还可以通过使用一个 `Condition` 对象来同步线程。由于 `Condition` 使用了一个 `Lock`，它可以绑定到一个共享资源，允许多个线程等待资源更新。在这个例子中，`consumer()` 线程要等待设置了 `Condition` 才能继续。`producer()` 线程负责设置条件，并通知其他线程可以继续。

```
import logging
import threading
import time

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s %(threadName)-2s %(message)s',
)

def consumer(cond):
    """wait for the condition and use the resource"""
    logging.debug('Starting consumer thread')
    t = threading.currentThread()
    with cond:
        cond.wait()
        logging.debug('Resource is available to consumer')

def producer(cond):
    """set up the resource to be used by the consumer"""
    logging.debug('Starting producer thread')
    with cond:
        logging.debug('Making resource available')
        cond.notifyAll()

condition = threading.Condition()
c1 = threading.Thread(name='c1', target=consumer,
                      args=(condition,))
c2 = threading.Thread(name='c2', target=consumer,
                      args=(condition,))
p = threading.Thread(name='p', target=producer,
                    args=(condition,))

c1.start()
```

```

time.sleep(2)
c2.start()
time.sleep(2)
p.start()

```

这些线程使用 `with` 来获得与 `Condition` 关联的锁。也可以显式地使用 `acquire()` 和 `release()` 方法。

```
$ python threading_condition.py
```

```

2010-11-15 09:24:53,544 (c1) Starting consumer thread
2010-11-15 09:24:55,545 (c2) Starting consumer thread
2010-11-15 09:24:57,546 (p ) Starting producer thread
2010-11-15 09:24:57,546 (p ) Making resource available
2010-11-15 09:24:57,547 (c2) Resource is available to consumer
2010-11-15 09:24:57,547 (c1) Resource is available to consumer

```

10.3.10 限制资源的并发访问

有时可能需要允许多个工作线程同时访问一个资源，但要限制总数。例如，连接池支持同时连接，但数目可能是固定的，或者一个网络应用可能支持固定数目的并发下载。这些连接就可以使用 `Semaphore` 来管理。

```

import logging
import random
import threading
import time

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s %(threadName)-2s %(message)s',
)

class ActivePool(object):
    def __init__(self):
        super(ActivePool, self).__init__()
        self.active = []
        self.lock = threading.Lock()
    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
            logging.debug('Running: %s', self.active)
    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
            logging.debug('Running: %s', self.active)

def worker(s, pool):
    logging.debug('Waiting to join the pool')

```




```

        val = data.value
    except AttributeError:
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)
def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)

local_data = threading.local()
show_value(local_data)
local_data.value = 1000
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()

```

属性 `local_data.value` 对所有线程都不可见，除非它在某个线程中设置才能被该线程看到。

\$ python threading_local.py

```

(MainThread) No value yet
(MainThread) value=1000
(Thread-1 ) No value yet
(Thread-1 ) value=71
(Thread-2 ) No value yet
(Thread-2 ) value=38

```

要初始化设置，使所有线程开始时都有相同的值，可以使用一个子类，并在 `__init__()` 中设置这些属性。

```

import random
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

def show_value(data):
    try:
        val = data.value
    except AttributeError:
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)

```




```

def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)

class MyLocal(threading.local):
    def __init__(self, value):
        logging.debug('Initializing %r', self)
        self.value = value

local_data = MyLocal(1000)
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()

```

这会在相同的对象上调用 `__init__()` (注意 `id()` 值), 每个线程中调用一次来设置默认值。

```
$ python threading_local_defaults.py
```

```

(MainThread) Initializing <__main__.MyLocal object at 0x100e16050>
(MainThread) value=1000
(Thread-1 ) Initializing <__main__.MyLocal object at 0x100e16050>
(Thread-1 ) value=1000
(Thread-1 ) value=19
(Thread-2 ) Initializing <__main__.MyLocal object at 0x100e16050>
(Thread-2 ) value=1000
(Thread-2 ) value=55

```

参见:

`threading` (<http://docs.python.org/lib/module-threading.html>) 这个模块的标准库文档。

`thread` 底层线程 API。

`multiprocessing` (10.4 节) 处理进程的一个 API; 它是 `threading` API 的镜像。

`Queue` (2.5 节) 这是一个线程安全队列, 可用来在线程之间传递消息。

10.4 multiprocessing——像线程一样管理进程

作用: 提供一个 API 来管理进程。

Python 版本: 2.6 及以后版本

`multiprocessing` 模块包含一个 API, 它基于 `threading` API 可以在多个进程间划分工作。有些情况下, `multiprocessing` 可以作为临时替换, 取代 `threading` 来利用多个 CPU 内核, 避免 Python 全局解释器锁所带来的计算瓶颈。

由于这种类似性，这里的前几个例子都由 threading 例子修改得来。multiprocessing 中有而 threading 未提供的特性将在后面介绍。

10.4.1 multiprocessing 基础

要创建第二个进程，最简单的方法是用一个目标函数实例化一个 Process 对象，并调用 start() 让它开始工作。

```
import multiprocessing

def worker():
    """worker function"""
    print 'Worker'
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker)
        jobs.append(p)
        p.start()
```

输出中单词“Worker”将打印 5 次，不过不能清楚地看出孰先孰后，这取决于具体的执行顺序，因为每个进程都在竞争访问输出流。

```
$ python multiprocessing_simple.py
```

```
Worker
Worker
Worker
Worker
Worker
```

更有用的做法是，创建一个进程时可以提供参数来告诉它要做什么。与 threading 不同，要向一个 multiprocessing Process 传递参数，这个参数必须能够使用 pickle 串行化。下面这个例子向各个工作进程传递一个要打印的数。

```
import multiprocessing

def worker(num):
    """thread worker function"""
    print 'Worker:', num
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
```

```
jobs.append(p)
p.start()
```

现在整数参数会包含在各个工作进程打印的消息中：

```
$ python multiprocessing_simpleargs.py
```

```
Worker: 0
Worker: 1
Worker: 4
Worker: 2
Worker: 3
```

10.4.2 可导入的目标函数

threading 与 multiprocessing 例子之间有一个区别，multiprocessing 例子中对 `__main__` 使用了额外的保护。由于新进程启动的方式，要求子进程能够导入包含目标函数的脚本。可以将应用的主要部分包装在一个 `__main__` 检查中，确保模块导入时不会在各个子进程中递归地运行。另一种方法是从一个单独的脚本导入目标函数。例如，`multiprocessing_import_main.py` 使用了第二个模块中定义的一个工作函数。

```
import multiprocessing
import multiprocessing_import_worker

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(
            target=multiprocessing_import_worker.worker,
        )
        jobs.append(p)
        p.start()
```

这个工作函数在 `multiprocessing_import_worker.py` 中定义。

```
def worker():
    """worker function"""
    print 'Worker'
    return
```

调用主程序会生成与第一个例子类似的输出。

```
$ python multiprocessing_import_main.py
```

```
Worker
Worker
Worker
Worker
Worker
```



10.4.3 确定当前进程

传递参数来标识或命名进程很麻烦，也没有必要。每个 Process 实例都有一个名称，其默认值可以在创建进程时改变。给进程命名对于跟踪进程很有用，特别是当应用中有多种类型的进程在同时运行时。

```
import multiprocessing
import time

def worker():
    name = multiprocessing.current_process().name
    print name, 'Starting'
    time.sleep(2)
    print name, 'Exiting'

def my_service():
    name = multiprocessing.current_process().name
    print name, 'Starting'
    time.sleep(3)
    print name, 'Exiting'

if __name__ == '__main__':
    service = multiprocessing.Process(name='my_service',
                                     target=my_service)
    worker_1 = multiprocessing.Process(name='worker 1',
                                     target=worker)
    worker_2 = multiprocessing.Process(target=worker) # default name

    worker_1.start()
    worker_2.start()
    service.start()
```

调试输出中，每行都包含当前进程的名称。进程名称列为 Process-3 的行对应未命名的进程 worker_1。

```
$ python multiprocessing_names.py
```

```
worker 1 Starting
worker 1 Exiting
Process-3 Starting
Process-3 Exiting
my_service Starting
my_service Exiting
```

10.4.4 守护进程

默认情况下，在所有子进程退出之前主程序不会退出。有些情况下，可能需要启动一个后台进程，它可以一直运行而不阻塞主程序退出，如果一个服务无法用一种容易的方法来中断进

程，或者希望进程工作到一半时中止而不损失或破坏数据（如为一个服务监控工具生成“心跳”的任务），对于这些服务，使用守护进程就很有用。

要标志一个进程为守护进程，可以将其 `daemon` 属性设置为 `True`。默认情况下进程不作为守护进程。

```
import multiprocessing
import time
import sys

def daemon():
    p = multiprocessing.current_process()
    print 'Starting:', p.name, p.pid
    sys.stdout.flush()
    time.sleep(2)
    print 'Exiting :', p.name, p.pid
    sys.stdout.flush()

def non_daemon():
    p = multiprocessing.current_process()
    print 'Starting:', p.name, p.pid
    sys.stdout.flush()
    print 'Exiting :', p.name, p.pid
    sys.stdout.flush()

if __name__ == '__main__':
    d = multiprocessing.Process(name='daemon', target=daemon)
    d.daemon = True

    n = multiprocessing.Process(name='non-daemon', target=non_daemon)
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()
```

输出中没有守护进程的“Exiting”消息，因为在守护进程从其 2 秒的睡眠时间唤醒之前，所有非守护进程（包括主程序）已经退出。

```
$ python multiprocessing_daemon.py
```

```
Starting: daemon 9842
Starting: non-daemon 9843
Exiting : non-daemon 9843
```

守护进程会在主程序退出之前自动终止，以避免留下“孤”进程继续运行。要验证这一点，可以查找程序运行时打印的进程 id 值，然后用一个类似 `ps` 的命令检查该进程。

10.4.5 等待进程

要等待一个进程完成工作并退出，可以使用 `join()` 方法。

```
import multiprocessing
import time
import sys

def daemon():
    name = multiprocessing.current_process().name
    print 'Starting:', name
    time.sleep(2)
    print 'Exiting :', name

def non_daemon():
    name = multiprocessing.current_process().name
    print 'Starting:', name
    print 'Exiting :', name

if __name__ == '__main__':
    d = multiprocessing.Process(name='daemon',
                                target=daemon)

    d.daemon = True

    n = multiprocessing.Process(name='non-daemon',
                                target=non_daemon)

    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()

    d.join()
    n.join()
```

由于主进程使用 `join()` 等待守护进程退出，所以这一次会打印 “Exiting” 消息。

```
$ python multiprocessing_daemon_join.py
```

```
Starting: non-daemon
Exiting : non-daemon
Starting: daemon
Exiting : daemon
```

默认情况下，`join()` 会无限阻塞。可以传入一个超时参数（这是一个浮点数，表示在进程变为不活动之前所等待的秒数）。即使进程在这个超时期限内没有完成，`join()` 也会返回。

```
import multiprocessing
import time
```

```

import sys

def daemon():
    name = multiprocessing.current_process().name
    print 'Starting:', name
    time.sleep(2)
    print 'Exiting :', name

def non_daemon():
    name = multiprocessing.current_process().name
    print 'Starting:', name
    print 'Exiting :', name

if __name__ == '__main__':
    d = multiprocessing.Process(name='daemon',
                                target=daemon)
    d.daemon = True

    n = multiprocessing.Process(name='non-daemon',
                                target=non_daemon)
    n.daemon = False

    d.start()
    n.start()
    d.join(1)
    print 'd.is_alive()', d.is_alive()
    n.join()

```

由于传入的超时值小于守护进程睡眠的时间，所以 join() 返回之后这个进程仍“存活”。

```
$ python multiprocessing_daemon_join_timeout.py
```

```

Starting: non-daemon
Exiting : non-daemon
d.is_alive() True

```

10.4.6 终止进程

尽管最好使用“毒丸”(poison pill)方法向进程发出信号告诉它应当退出(见本章 10.4.10 节)，但是如果一个进程看起来已经挂起或陷入死锁，则需要能够强制性地将其结束。对一个进程对象调用 terminate() 会结束子进程。

```

import multiprocessing
import time

def slow_worker():
    print 'Starting worker'
    time.sleep(0.1)
    print 'Finished worker'

```

```

if __name__ == '__main__':
    p = multiprocessing.Process(target=slow_worker)
    print 'BEFORE:', p, p.is_alive()

    p.start()
    print 'DURING:', p, p.is_alive()

    p.terminate()
    print 'TERMINATED:', p, p.is_alive()

    p.join()
    print 'JOINED:', p, p.is_alive()

```

注意：终止进程后要使用 `join()` 退出进程，使进程管理代码有时间更新对象的状态，以反映进程已经终止。

```
$ python multiprocessing_terminate.py
```

```

BEFORE: <Process(Process-1, initial)> False
DURING: <Process(Process-1, started)> True
TERMINATED: <Process(Process-1, started)> True
JOINED: <Process(Process-1, stopped[SIGTERM])> False

```

10.4.7 进程退出状态

进程退出时生成的状态码可以通过 `exitcode` 属性访问。表 10.1 列出了可用的退出码范围。

表 10.1 Multiprocessing 退出码

退 出 码	含 义
<code>== 0</code>	未生成任何错误
<code>> 0</code>	进程有一个错误，并以该错误码退出
<code>< 0</code>	进程由一个 <code>-1 * exitcode</code> 信号结束

```

import multiprocessing
import sys
import time

def exit_error():
    sys.exit(1)

def exit_ok():
    return

def return_value():

```



```

    return 1

def raises():
    raise RuntimeError('There was an error!')
def terminated():
    time.sleep(3)

if __name__ == '__main__':
    jobs = []
    for f in [exit_error, exit_ok, return_value, raises, terminated]:
        print 'Starting process for', f.func_name
        j = multiprocessing.Process(target=f, name=f.func_name)
        jobs.append(j)
        j.start()

    jobs[-1].terminate()

    for j in jobs:
        j.join()
        print '%15s.exitcode = %s' % (j.name, j.exitcode)

```

产生异常的进程会自动得到 exitcode 为 1。

```
$ python multiprocessing_exitcode.py
```

```

Starting process for exit_error
Starting process for exit_ok
Starting process for return_value
Starting process for raises
Starting process for terminated
Process raises:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/multiprocessing/process.py", line 232, in _bootstrap
    self.run()
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/multiprocessing/process.py", line 88, in run
    self._target(*self._args, **self._kwargs)
  File "multiprocessing_exitcode.py", line 24, in raises
    raise RuntimeError('There was an error!')
RuntimeError: There was an error!
    exit_error.exitcode = 1
    exit_ok.exitcode = 0
    return_value.exitcode = 0
    raises.exitcode = 1
    terminated.exitcode = -15

```

10.4.8 日志

调试并发问题时，能够访问 `multiprocessing` 提供的对象的内部状态会很有用。可以使用一个方便的模块级函数来启用日志记录，名为 `log_to_stderr()`。它使用 `logging` 建立一个日志记录器对象，并增加一个处理程序，使得日志消息将发送到标准错误通道。

```
import multiprocessing
import logging
import sys

def worker():
    print 'Doing some work'
    sys.stdout.flush()

if __name__ == '__main__':
    multiprocessing.log_to_stderr(logging.DEBUG)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

默认情况下，日志级别设置为 `NOTSET`，即不产生任何消息。通过传入一个不同的日志级别，可以初始化日志记录器，指定所需的详细程度。

```
$ python multiprocessing_log_to_stderr.py
```

```
[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[DEBUG/Process-1] running all "atexit" finalizers with priority >= 0
[DEBUG/Process-1] running the remaining "atexit" finalizers
[INFO/Process-1] process exiting with exitcode 0
[INFO/MainProcess] process shutting down
[DEBUG/MainProcess] running all "atexit" finalizers with priority >= 0
[DEBUG/MainProcess] running the remaining "atexit" finalizers
```

要直接处理日志记录器（修改其日志级别或添加处理程序），可以使用 `get_logger()`。

```
import multiprocessing
import logging
import sys

def worker():
    print 'Doing some work'
    sys.stdout.flush()

if __name__ == '__main__':
    multiprocessing.log_to_stderr()
    logger = multiprocessing.get_logger()
    logger.setLevel(logging.INFO)
```

```
p = multiprocessing.Process(target=worker)
p.start()
p.join()
```

使用名称 `multiprocessing`，还可以通过 `logging` 配置文件 API 来配置日志记录器。

```
$ python multiprocessing_get_logger.py
```

```
[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[INFO/Process-1] process exiting with exitcode 0
[INFO/MainProcess] process shutting down
```

10.4.9 派生进程

要在一个单独的进程中开始工作，尽管最简单的方法是使用 `Process` 并传入一个目标函数，不过也可以使用一个定制子类。

```
import multiprocessing

class Worker(multiprocessing.Process):

    def run(self):
        print 'In %s' % self.name
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = Worker()
        jobs.append(p)
        p.start()
    for j in jobs:
        j.join()
```

派生类应当覆盖 `run()` 来完成工作。

```
$ python multiprocessing_subclass.py
```

```
In Worker-1
In Worker-2
In Worker-3
In Worker-4
In Worker-5
```

10.4.10 向进程传递消息

类似于线程，对于多个进程，一种常用的模式是将一个工作划分到多个工作进程中并行地运行。要想有效地使用多个进程，通常要求它们之间有某种通信，这样才能分解工作，并完成

结果的汇总。利用 multiprocessing 完成进程间通信的一种简单方法是使用一个 Queue 来回传递消息。能够用 pickle 串行化的任何对象都可以通过 Queue 传递。

```
import multiprocessing

class MyFancyClass(object):

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print 'Doing something fancy in %s for %s!' % \
            (proc_name, self.name)

def worker(q):
    obj = q.get()
    obj.do_something()

if __name__ == '__main__':
    queue = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

这个小例子只是向一个工作进程传递一个消息，然后主进程等待这个工作进程完成。

```
$ python multiprocessing_queue.py
```

```
Doing something fancy in Process-1 for Fancy Dan!
```

来看一个更复杂的例子，这里展示了如何管理多个工作进程，它们都利用一个 JoinableQueue 的数据，并把结果传递回父进程。这里使用“毒丸”技术来停止工作进程。建立具体任务后，主程序会在作业队列中为每个工作进程添加一个“stop”值。当一个工作进程遇到这个特定值时，就会退出其处理循环。主进程使用任务队列的 join() 方法等待所有任务都完成后才开始处理结果。

```
import multiprocessing
import time

class Consumer(multiprocessing.Process):
```

```

def __init__(self, task_queue, result_queue):
    multiprocessing.Process.__init__(self)
    self.task_queue = task_queue
    self.result_queue = result_queue

def run(self):
    proc_name = self.name
    while True:
        next_task = self.task_queue.get()
        if next_task is None:
            # Poison pill means shutdown
            print '%s: Exiting' % proc_name
            self.task_queue.task_done()
            break
        print '%s: %s' % (proc_name, next_task)
        answer = next_task()
        self.task_queue.task_done()
        self.result_queue.put(answer)
    return

class Task(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __call__(self):
        time.sleep(0.1) # pretend to take some time to do the work
        return '%s * %s = %s' % (self.a, self.b, self.a * self.b)
    def __str__(self):
        return '%s * %s' % (self.a, self.b)

if __name__ == '__main__':
    # Establish communication queues
    tasks = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()

    # Start consumers
    num_consumers = multiprocessing.cpu_count() * 2
    print 'Creating %d consumers' % num_consumers
    consumers = [ Consumer(tasks, results)
                  for i in xrange(num_consumers) ]
    for w in consumers:
        w.start()

    # Enqueue jobs
    num_jobs = 10

```

```
for i in xrange(num_jobs):
    tasks.put(Task(i, i))

# Add a poison pill for each consumer
for i in xrange(num_consumers):
    tasks.put(None)
# Wait for all the tasks to finish
tasks.join()

# Start printing results
while num_jobs:
    result = results.get()
    print 'Result:', result
    num_jobs -= 1
```

尽管作业按顺序进入队列，但它们的执行却是并行的，所以不能保证它们完成的顺序。

```
$ python -u multiprocessing_producer_consumer.py
```

```
Creating 4 consumers
Consumer-1: 0 * 0
Consumer-2: 1 * 1
Consumer-3: 2 * 2
Consumer-4: 3 * 3
Consumer-4: 4 * 4
Consumer-1: 5 * 5
Consumer-3: 6 * 6
Consumer-2: 7 * 7
Consumer-1: 8 * 8
Consumer-4: 9 * 9
Consumer-3: Exiting
Consumer-2: Exiting
Consumer-1: Exiting
Consumer-4: Exiting
Result: 0 * 0 = 0
Result: 3 * 3 = 9
Result: 2 * 2 = 4
Result: 1 * 1 = 1
Result: 5 * 5 = 25
Result: 4 * 4 = 16
Result: 6 * 6 = 36
Result: 7 * 7 = 49
Result: 9 * 9 = 81
Result: 8 * 8 = 64
```

10.4.11 进程间信号传输

Event 类提供了一种简单的方法，可以在进程之间传递状态信息。事件可以切换设置和未



设置状态。通过使用一个可选的超时值，事件对象的用户可以等待其状态从未设置变为设置。

```
import multiprocessing
import time

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print 'wait_for_event: starting'
    e.wait()
    print 'wait_for_event: e.is_set()->', e.is_set()

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    print 'wait_for_event_timeout: starting'
    e.wait(t)
    print 'wait_for_event_timeout: e.is_set()->', e.is_set()

if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(name='block',
                                target=wait_for_event,
                                args=(e,))
    w1.start()

    w2 = multiprocessing.Process(name='nonblock',
                                target=wait_for_event_timeout,
                                args=(e, 2))
    w2.start()

    print 'main: waiting before calling Event.set()'
    time.sleep(3)
    e.set()
    print 'main: event is set'
```

wait() 到时间时就会返回，而没有任何错误。调用者负责使用 is_set() 检查事件的状态。

```
$ python -u multiprocessing_event.py
```

```
main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is setwait_for_event: e.is_set()->
True
```

10.4.12 控制资源访问

如果需要在多个进程间共享一个资源，在这种情况下，可以使用一个 Lock 来避免访问冲突。

```

import multiprocessing
import sys

def worker_with(lock, stream):
    with lock:
        stream.write('Lock acquired via with\n')

def worker_no_with(lock, stream):
    lock.acquire()
    try:
        stream.write('Lock acquired directly\n')
    finally:
        lock.release()

lock = multiprocessing.Lock()
w = multiprocessing.Process(target=worker_with,
                             args=(lock, sys.stdout))
nw = multiprocessing.Process(target=worker_no_with,
                              args=(lock, sys.stdout))

w.start()
nw.start()

w.join()
nw.join()

```

在这个例子中，如果这两个进程没有用锁同步其输出流访问，打印到控制台的消息可能会纠缠在一起。

```
$ python multiprocessing_lock.py
```

```

Lock acquired via with
Lock acquired directly

```

10.4.13 同步操作

Condition 对象可以用来同步一个工作流的各个部分，使其中一些部分并行运行，而另外一些顺序运行，即使它们在单独的进程中。

```

import multiprocessing
import time

def stage_1(cond):
    """perform first stage of work,
    then notify stage_2 to continue
    """
    name = multiprocessing.current_process().name
    print 'Starting', name

```



```

with cond:
    print '%s done and ready for stage 2' % name
    cond.notify_all()

def stage_2(cond):
    """wait for the condition telling us stage_1 is done"""
    name = multiprocessing.current_process().name
    print 'Starting', name
    with cond:
        cond.wait()
        print '%s running' % name

if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='s1',
                                target=stage_1,
                                args=(condition,))

    s2_clients = [
        multiprocessing.Process(name='stage_2[%d]' % i,
                                target=stage_2,
                                args=(condition,))
        for i in range(1, 3)
    ]
    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()

```

这个例子中，两个进程并行地运行一个作业的第二阶段，但前提是第一阶段已经完成。

```
$ python multiprocessing_condition.py
```

```

Starting s1
s1 done and ready for stage 2
Starting stage_2[1]
stage_2[1] running
Starting stage_2[2]
stage_2[2] running

```

10.4.14 控制资源的并发访问

有时可能需要允许多个工作进程同时访问一个资源，但要限制总数。例如，连接池支持同时连接，但数目可能是固定的，或者一个网络应用可能支持固定数目的并发下载。这些连接就

可以使用 Semaphore 来管理。

```
import random
import multiprocessing
import time

class ActivePool(object):
    def __init__(self):
        super(ActivePool, self).__init__()
        self.mgr = multiprocessing.Manager()
        self.active = self.mgr.list()
        self.lock = multiprocessing.Lock()
    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
    def __str__(self):
        with self.lock:
            return str(self.active)

def worker(s, pool):
    name = multiprocessing.current_process().name
    with s:
        pool.makeActive(name)
        print 'Now running: %s' % str(pool)
        time.sleep(random.random())
        pool.makeInactive(name)

if __name__ == '__main__':
    pool = ActivePool()
    s = multiprocessing.Semaphore(3)
    jobs = [
        multiprocessing.Process(target=worker,
                                name=str(i),
                                args=(s, pool),
                                )
        for i in range(10)
    ]

    for j in jobs:
        j.start()

    for j in jobs:
        j.join()
    print 'Now running: %s' % str(pool)
```



在这个例子中，ActivePool 类只作为一种便利方法，用来跟踪某个给定时刻哪些进程能够运行。真正的资源池会为新的活动进程分配一个连接或另外某个值，这个进程工作完成时再回收这个值。在这里，资源池只是用来保存活动进程的名称，以显示至少有三个进程在并发运行。

```
$ python multiprocessing_semaphore.py
```

```
Now running: ['0', '1', '3']
Now running: ['0', '1', '3']
Now running: ['3', '2', '5']
Now running: ['0', '1', '3']
Now running: ['1', '3', '2']
Now running: ['2', '6', '7']
Now running: ['3', '2', '6']
Now running: ['6', '4', '8']
Now running: ['4', '8', '9']
Now running: ['6', '7', '4']
Now running: ['1', '3', '2']
Now running: ['3', '2', '5']
Now running: ['6', '7', '4']
Now running: ['6', '7', '4']
Now running: []
Now running: []
Now running: []
Now running: []
Now running: []
Now running: []
```

10.4.15 管理共享状态

在前面的例子中，ActivePool 实例通过一个由 Manager 创建的特殊类型列表对象集中维护活动进程列表。Manager 负责协调其所有用户之间的共享信息状态。

```
import multiprocessing
import pprint

def worker(d, key, value):
    d[key] = value

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    d = mgr.dict()
    jobs = [ multiprocessing.Process(target=worker, args=(d, i, i*2))
             for i in range(10) ]
    for j in jobs:
        j.start()
    for j in jobs:
```

```
j.join()
print 'Results:', d
```

通过管理器来创建列表，这个列表将会共享，所有进程都能看到列表更新。除了列表，管理器还支持字典。

```
$ python multiprocessing_manager_dict.py
```

```
Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14,
8: 16, 9: 18}
```

10.4.16 共享命名空间

除了字典和列表之外，Manager 还可以创建一个共享 Namespace。

```
import multiprocessing

def producer(ns, event):
    ns.value = 'This is the value'
    event.set()

def consumer(ns, event):
    try:
        value = ns.value
    except Exception, err:
        print 'Before event, error:', str(err)
    event.wait()
    print 'After event:', ns.value

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    event = multiprocessing.Event()
    p = multiprocessing.Process(target=producer,
                                args=(namespace, event))
    c = multiprocessing.Process(target=consumer,
                                args=(namespace, event))

    c.start()
    p.start()
    c.join()
    p.join()
```

添加到 Namespace 的所有命名值对所有接收 Namespace 实例的客户都可见。

```
$ python multiprocessing_namespaces.py
```

```
Before event, error: 'Namespace' object has no attribute 'value'
After event: This is the value
```

要知道重要的一点，对命名空间中可变值内容的更新不会自动传播。

```
import multiprocessing

def producer(ns, event):
    # DOES NOT UPDATE GLOBAL VALUE!
    ns.my_list.append('This is the value')
    event.set()

def consumer(ns, event):
    print 'Before event:', ns.my_list
    event.wait()
    print 'After event :', ns.my_list

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    namespace.my_list = []

    event = multiprocessing.Event()
    p = multiprocessing.Process(target=producer,
                               args=(namespace, event))
    c = multiprocessing.Process(target=consumer,
                               args=(namespace, event))

    c.start()
    p.start()

    c.join()
    p.join()
```

要更新这个列表，需要将它再次关联到命名空间对象。

```
$ python multiprocessing_namespaces_mutable.py
```

```
Before event: []
After event : []
```

10.4.17 进程池

有些情况下，所要完成的工作可以分解并独立地分布到多个工作进程，对于这种简单的情况，可以用 Pool 类来管理固定数目的工作进程。作业的返回值会收集并作为一个列表返回。池 (pool) 参数包括进程数以及启动任务进程时要运行的函数（对每个子进程调用一次）。

```
import multiprocessing

def do_calculation(data):
    return data * 2
```

```

def start_process():
    print 'Starting', multiprocessing.current_process().name

if __name__ == '__main__':
    inputs = list(range(10))
    print 'Input      :', inputs

    builtin_outputs = map(do_calculation, inputs)
    print 'Built-in:', builtin_outputs

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process,
                                )

    pool_outputs = pool.map(do_calculation, inputs)
    pool.close() # no more tasks
    pool.join()  # wrap up current tasks

    print 'Pool      :', pool_outputs

```

map() 方法在功能上等价于内置 map(), 只不过单个任务会并行运行。由于进程池并行地处理输入, 可以用 close() 和 join() 使任务进程与主进程同步, 以确保完成适当的清理。

```
$ python multiprocessing_pool.py
```

```

Input      : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Starting PoolWorker-3
Starting PoolWorker-1
Starting PoolWorker-4
Starting PoolWorker-2
Pool      : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

```

默认情况下, Pool 会创建固定数目的工作进程, 并向这些工作进程传递作业, 直到再没有更多作业为止。设置 maxtasksperchild 参数可以告诉池在完成一些任务之后要重新启动一个工作进程, 来避免运行时间很长的工作进程消耗太多的系统资源。

```

import multiprocessing

def do_calculation(data):
    return data * 2

def start_process():
    print 'Starting', multiprocessing.current_process().name

if __name__ == '__main__':
    inputs = list(range(10))
    print 'Input      :', inputs

```

```

builtin_outputs = map(do_calculation, inputs)
print 'Built-in:', builtin_outputs

pool_size = multiprocessing.cpu_count() * 2
pool = multiprocessing.Pool(processes=pool_size,
                             initializer=start_process,
                             maxtasksperchild=2,
                             )
pool_outputs = pool.map(do_calculation, inputs)
pool.close() # no more tasks
pool.join()  # wrap up current tasks

print 'Pool      :', pool_outputs

```

池完成其所分配的任务时，即使并没有更多工作要做，也会重新启动工作进程。从这个输出可以看到，尽管只有 10 个任务，而且每个工作进程一次可以完成两个任务，但是这里创建了 8 个工作进程。

```
$ python multiprocessing_pool_maxtasksperchild.py
```

```

Input      : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Starting PoolWorker-1
Starting PoolWorker-2
Starting PoolWorker-3
Starting PoolWorker-4
Starting PoolWorker-5
Starting PoolWorker-6
Starting PoolWorker-7
Starting PoolWorker-8
Pool       : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

```

10.4.18 实现 MapReduce

Pool 类可以用于创建一个简单的单服务器 MapReduce 实现。尽管它无法充分提供分布处理的好处，但确实展示了将一些问题分解到可分布的工作单元是何等容易。

在基于 MapReduce 的系统中，输入数据分解为块，由不同的工作进程实例处理。首先使用一个简单的转换将各个输入数据块映射到一个中间状态。然后将中间数据汇集在一起，基于一个键值分区，使所有相关的值都在一起。最后，将分区的数据归约为一个结果集。

```

import collections
import itertools
import multiprocessing

class SimpleMapReduce(object):

```

```
def __init__(self, map_func, reduce_func, num_workers=None):
    """
    map_func
    Function to map inputs to intermediate data. Takes as
    argument one input value and returns a tuple with the key
    and a value to be reduced.

    reduce_func

    Function to reduce partitioned version of intermediate data
    to final output. Takes as argument a key as produced by
    map_func and a sequence of the values associated with that
    key.

    num_workers

    The number of workers to create in the pool. Defaults to
    the number of CPUs available on the current host.
    """
    self.map_func = map_func
    self.reduce_func = reduce_func
    self.pool = multiprocessing.Pool(num_workers)

def partition(self, mapped_values):
    """Organize the mapped values by their key.
    Returns an unsorted sequence of tuples with a key
    and a sequence of values.
    """
    partitioned_data = collections.defaultdict(list)
    for key, value in mapped_values:
        partitioned_data[key].append(value)
    return partitioned_data.items()

def __call__(self, inputs, chunksize=1):
    """Process the inputs through the map and reduce functions
    given.

    inputs
    An iterable containing the input data to be processed.

    chunksize=1
    The portion of the input data to hand to each worker. This
    can be used to tune performance during the mapping phase.
    """
    map_responses = self.pool.map(self.map_func,
                                  inputs,
                                  chunksize=chunksize)
```



```

partitioned_data = self.partition(
    itertools.chain(*map_responses)
)
reduced_values = self.pool.map(self.reduce_func,
                                partitioned_data)

return reduced_values

```

以下示例脚本使用 SimpleMapReduce 统计这篇文章 reStructuredText 源中的“单词”数，这里要忽略其中的一些标记。

```

import multiprocessing
import string

from multiprocessing_mapreduce import SimpleMapReduce

def file_to_words(filename):
    """Read a file and return a sequence of
    (word, occurrences) values.
    """
    STOP_WORDS = set([
        'a', 'an', 'and', 'are', 'as', 'be', 'by', 'for', 'if',
        'in', 'is', 'it', 'of', 'or', 'py', 'rst', 'that', 'the',
        'to', 'with',
    ])
    TR = string.maketrans(string.punctuation,
                           ' ' * len(string.punctuation))

    print multiprocessing.current_process().name, 'reading', filename
    output = []

    with open(filename, 'rt') as f:
        for line in f:
            if line.lstrip().startswith('.'): # Skip comment lines
                continue
            line = line.translate(TR) # Strip punctuation
            for word in line.split():
                word = word.lower()
                if word.isalpha() and word not in STOP_WORDS:
                    output.append( (word, 1) )
    return output

def count_words(item):
    """Convert the partitioned data for a word to a
    tuple containing the word and the number of occurrences.
    """
    word, occurrences = item
    return (word, sum(occurrences))

```

```

if __name__ == '__main__':
    import operator
    import glob

    input_files = glob.glob('*.rst')

    mapper = SimpleMapReduce(file_to_words, count_words)
    word_counts = mapper(input_files)
    word_counts.sort(key=operator.itemgetter(1))
    word_counts.reverse()

    print '\nTOP 20 WORDS BY FREQUENCY\n'
    top20 = word_counts[:20]
    longest = max(len(word) for word, count in top20)
    for word, count in top20:
        print '%-10s: %5s' % (longest+1, word, count)

```

`file_to_words()` 函数将各个输入文件转换为一个元组序列，各元组包含单词和数字 1（表示一次出现）。`partition()` 使用单词作为键来划分数据，所以得到的结构包括一个键和一个 1 值序列（表示单词的各次出现）。分区数据转换为一组元组，元组中包含一个单词和归约阶段中 `count_words()` 统计得出的这个单词的出现次数。

```
$ python multiprocessing_wordcount.py
```

```

PoolWorker-1 reading basics.rst
PoolWorker-1 reading index.rst
PoolWorker-2 reading communication.rst
PoolWorker-2 reading mapreduce.rst

```

```
TOP 20 WORDS BY FREQUENCY
```

```

process      :    81
multiprocessing :    43
worker       :    38
after        :    34
starting     :    33
running      :    32
processes    :    32
python       :    31
start        :    29
class        :    28
literal      :    27
header       :    27
pymotw       :    27
end          :    27
daemon       :    23
now          :    22

```



func	:	21
can	:	21
consumer	:	20
mod	:	19

参见:

`multiprocessing` (<http://docs.python.org/library/multiprocessing.html>) 这个模块的标准库文档。

`MapReduce` (<http://en.wikipedia.org/wiki/MapReduce>) 维基百科上关于 `MapReduce` 的概述。

`MapReduce: Simplified Data Processing on Large Clusters` (<http://labs.google.com/papers/mapreduce.html>) Google Labs 关于 `MapReduce` 的演示文稿和论文。

`operator` (3.3 节) 操作符工具如 `itemgetter()`。

`threading` (10.3 节) 处理线程的高级 API。



第 11 章

网络通信

网络通信用于获取一个算法在本地运行所需的数据，还可以共享信息实现分布式处理，另外可以用来管理云服务。Python 的标准库提供了一些模块来创建网络服务以及远程访问现有服务。

底层 socket 库允许直接访问本地 C 套接字库，可以用于与任何网络服务通信。select 同时监视多个套接字，允许网络服务器同时与多个客户通信。

SocketServer 中的框架抽象出创建一个新的网络服务器所需的大量重复性工作。可以结合这些类创建服务器来创建或使用线程，并支持 TCP 或 UDP。应用只需要提供具体的消息处理。

asyncore 实现了一个异步网络栈，并提供一个基于回调的 API。它封装了轮询循环和缓冲处理，接收到数据时会调用适当的处理程序。asynchat 中的框架在 asyncore 的基础上简化了创建基于双向消息的协议所需的工作。

11.1 socket——网络通信

作用：提供对网络通信的访问。

Python 版本：1.4 及以后版本

socket 模块提供了一个底层 C API，可以使用 BSD 套接字接口实现网络通信。它包括 socket 类，用于处理具体的数据通道，还包括用于完成网络相关任务的函数，如将一个服务器名称转换为一个地址，以及格式化数据以便在网络上发送。

11.1.1 寻址、协议簇和套接字类型

套接字 (socket) 是程序在本地或者通过互联网来回传递数据时所用通信通道的一个端点。套接字有两个主要属性来控制如何发送数据：地址簇 (address family) 控制所用的 OSI 网络层协议，套接字类型 (socket type) 控制传输层协议。

Python 支持 3 个地址簇。最常用的是 AF_INET，用于 IPv4 Internet 寻址。IPv4 地址长度为 4 个字节，通常表示为 4 个数的序列，每个字节对应一个数，用点号分隔（如 10.1.1.5 和 127.0.0.1）。这些值通常称为“IP 地址”。目前几乎所有互联网网络应用都使用 IPv4。

AF_INET6 用于 IPv6 Internet 寻址。IPv6 是“下一代”Internet 协议。它支持 128 位地址和通信流调整，还支持 IPv4 所不支持的一些路由特性。目前采用 IPv6 的应用还很有限，不过在不断增长。

AF_UNIX 是 UNIX 域套接字 (UNIX Domain Sockets, UDS) 的地址簇，这是一种 POSIX

兼容系统上的进程间通信协议。UDS 的实现通常允许操作系统直接从进程向进程传递数据，而不用通过网络栈。这比使用 AF_INET 更高效，但是由于要用文件系统作为寻址的命名空间，UDS 仅限于同一个系统上的进程。相比其他 IPC 机制（如命名管道或共享内存），使用 UDS 的优势在于，它与 IP 网络应用的编程接口是一样的。这说明，应用在单个主机上运行时可以利用高效的通信，在网络上发送数据时仍然可以使用同样的代码。

注意：AF_UNIX 常量仅在支持 UDS 的系统上定义。

套接字类型往往是 SOCK_DGRAM 或 SOCK_STREAM，SOCK_DGRAM 对应用户数据报协议（user datagram protocol，UDP），SOCK_STREAM 对应传输控制协议（transmission control protocol，TCP）。UDP 不需要传输握手过程或其他设置过程（setup），但是提供的传输可靠性较低。UDP 消息可能乱序传送，也可能传送多次，或者根本不传送。TCP 则相反，可以确保每个消息只传送一次，而且按正确的顺序传送。不过，由于增加了可靠性，可能会引入额外的延迟，因为数据包可能需要重新传输。大多数传送大量数据的应用协议（如 HTTP）都建立在 TCP 基础上。UDP 通常用于顺序不太重要的协议（因为消息可以放在一个数据包中，例如 DNS），或者用于广播（向多个主机发送相同的数据）。

注意：Python 的 socket 模块还支持其他套接字类型，不过它们不太常用，所以这里不做介绍。有关的更多详细信息可以参考标准库文档。

在网络上查找主机

socket 包含一些函数与网络上的域名服务交互，使得程序可以将服务器的主机名转换为其数字网络地址。应用使用地址来连接一个服务器之前并不需要显式地转换地址，不过报告错误时除了报告所用的名字之外，还包含这个数字地址会很有用。

要查找当前主机的正式名字，可以使用 gethostname()。

```
import socket

print socket.gethostname()
```

所返回的名字取决于当前系统的网络设置，在不同的网络上返回的名字可能有变化（如一个连接到 WLAN 的笔记本电脑）。

```
$ python socket_gethostname.py

farnsworth.hellfly.net
```

这里使用 gethostbyname() 访问操作系统主机名解析 API，将服务器名字转换为其数字地址。

```
import socket

for host in [ 'homer', 'www', 'www.python.org', 'nosuchname' ]:
    try:
```

```

    print '%s : %s' % (host, socket.gethostbyname(host))
except socket.error, msg:
    print '%s : %s' % (host, msg)

```

如果当前系统的 DNS 配置在搜索中包括一个或多个域，名字 (name) 参数不要求是完全限定名 (也就是说，不需要包含域名以及基名)。如果一个名字无法找到，会产生一个 socket.error 类型的异常。

```
$ python socket_gethostbyname.py
```

```

homer : 192.168.1.8
www : 192.168.1.8
www.python.org : 82.94.164.162
nosuchname : [Errno 8] nodename nor servname provided, or not known

```

要访问有关服务器的更多命名信息，可以使用函数 `gethostbyname_ex()`。它会返回服务器的标准主机名、所有别名，以及可以用来到达这个主机的所有可用 IP 地址。

```

import socket

for host in [ 'homer', 'www', 'www.python.org', 'nosuchname' ]:
    print host
    try:
        hostname, aliases, addresses = socket.gethostbyname_ex(host)
        print '  Hostname:', hostname
        print '  Aliases :', aliases
        print '  Addresses:', addresses
    except socket.error as msg:
        print 'ERROR:', msg
    print

```

如果能得到一个服务器的所有已知 IP 地址，客户就可以实现其自己的负载平衡或故障恢复算法。

```
$ python socket_gethostbyname_ex.py
```

```

homer
  Hostname: homer.hellfly.net
  Aliases : []
  Addresses: ['192.168.1.8']

www
  Hostname: homer.hellfly.net
  Aliases : ['www.hellfly.net']
  Addresses: ['192.168.1.8']

www.python.org
  Hostname: www.python.org

```



```
Aliases : []
Addresses: ['82.94.164.162']
```

```
nosuchname
```

```
ERROR: [Errno 8] nodename nor servname provided, or not known
```

使用 `getfqdn()` 可以将一个部分名转换为完全限定域名。

```
import socket
```

```
for host in [ 'homer', 'www' ]:
    print '%6s : %s' % (host, socket.getfqdn(host))
```

如果输入是一个别名（如这里的 `www`），返回的名字不一定与输入参数一致。

```
$ python socket_getfqdn.py
```

```
homer : homer.hellfly.net
www : homer.hellfly.net
```

如果得到一个服务器的地址，可以使用 `gethostbyaddr()` 完成一个“逆向”查找来得到主机名。

```
import socket
```

```
hostname, aliases, addresses = socket.gethostbyaddr('192.168.1.8')
```

```
print 'Hostname :', hostname
print 'Aliases :', aliases
print 'Addresses:', addresses
```

返回值是一个元组，其中包含完全主机名、所有别名，以及与这个名字关联的所有 IP 地址。

```
$ python socket_gethostbyaddr.py
```

```
Hostname : homer.hellfly.net
Aliases : ['8.1.168.192.in-addr.arpa']
Addresses: ['192.168.1.8']
```

查找服务信息

除了 IP 地址之外，每个套接字地址还包括一个整数端口号（port number）。很多应用可以在同一个端口上运行并监听一个 IP 地址，不过一次只有一个套接字可以使用该地址的端口。通过结合 IP 地址、协议和端口号，可以惟一地标识一个通信通道，确保通过一个套接字发送的消息可以到达正确的目标。

有些端口号已经预先分配给某个特定协议。例如，使用 SMTP 的 email 服务器使用 TCP 在端口 25 相互通信，Web 客户和服务端使用 80 作为 HTTP 的端口号。网络服务的端口号和标准名可以使用 `getservbyname()` 查找。

```
import socket
from urlparse import urlparse
```

```

for url in [ 'http://www.python.org',
             'https://www.mybank.com',
             'ftp://prep.ai.mit.edu',
             'gopher://gopher.micro.umn.edu',
             'smtp://mail.example.com',
             'imap://mail.example.com',
             'imaps://mail.example.com',
             'pop3://pop.example.com',
             'pop3s://pop.example.com',
             ]:
    parsed_url = urlparse(url)
    port = socket.getservbyname(parsed_url.scheme)
    print '%6s : %s' % (parsed_url.scheme, port)

```

尽管标准化服务不太可能改变端口，不过最好用一个系统调用查找端口值，而不是在程序中硬编码写出端口号，这样在增加新服务时会更为灵活。

```
$ python socket_getservbyname.py
```

```

http : 80
https : 443
ftp : 21
gopher : 70
smtp : 25
imap : 143
imaps : 993
pop3 : 110
pop3s : 995

```

要逆向完成服务端口查找，可以使用 `getservbyport()`。

```

import socket
import urlparse

```

```

for port in [ 80, 443, 21, 70, 25, 143, 993, 110, 995 ]:
    print urlparse.urlunparse(
        (socket.getservbyport(port), 'example.com', '/', '', '', ''))

```

要从任意的地址构造服务 URL，这个逆向查找就很有用。

```
$ python socket_getservbyport.py
```

```

http://example.com/
https://example.com/
ftp://example.com/
gopher://example.com/
smtp://example.com/
imap://example.com/
imaps://example.com/

```



```
pop3://example.com/
pop3s://example.com/
```

可以使用 `getprotobyname()` 获取分配给一个传输协议的端口号。

```
import socket

def get_constants(prefix):
    """Create a dictionary mapping socket module
    constants to their names.
    """
    return dict( (getattr(socket, n), n)
                  for n in dir(socket)
                  if n.startswith(prefix)
                )

protocols = get_constants('IPPROTO_')
for name in [ 'icmp', 'udp', 'tcp' ]:
    proto_num = socket.getprotobyname(name)
    const_name = protocols[proto_num]
    print '%4s -> %2d (socket.%-12s = %2d)' % \
        (name, proto_num, const_name, getattr(socket, const_name))
```

协议码值是标准化的，作为常量在 `socket` 中定义，这些协议码都有前缀 `IPPROTO_`。

```
$ python socket_getprotobyname.py
```

```
icmp -> 1 (socket.IPPROTO_ICMP = 1)
udp -> 17 (socket.IPPROTO_UDP = 17)
tcp -> 6 (socket.IPPROTO_TCP = 6)
```

查找服务器地址

`getaddrinfo()` 将一个服务的基本地址转换为一个元组列表，其中包含建立一个连接所需的全部信息。每个元组的内容会有变化，包含不同的网络簇或协议。

```
import socket

def get_constants(prefix):
    """Create a dictionary mapping socket module
    constants to their names.
    """
    return dict( (getattr(socket, n), n)
                  for n in dir(socket)
                  if n.startswith(prefix)
                )

families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')
```




```

        socket.SOCK_STREAM, # socktype
        socket.IPPROTO_TCP, # protocol
        socket.AI_CANONNAME, # flags
    ):

    # Unpack the response tuple
    family, socktype, proto, canonname, sockaddr = response

    print 'Family      :', families[family]
    print 'Type        :', types[socktype]
    print 'Protocol     :', protocols[proto]
    print 'Canonical name:', canonname
    print 'Socket address:', sockaddr
    print

```

这一次由于标志 (flags) 包括 AI_CANONNAME, 如果这个主机有别名, 结果中会包含服务器的标准名 (可能与查找所用的值不同)。如果没有这个标志, 标准名值则仍为空。

```
$ python socket_getaddrinfo_extra_args.py
```

```

Family      : AF_INET
Type        : SOCK_STREAM
Protocol     : IPPROTO_TCP
Canonical name: homer.doughellmann.com
Socket address: ('192.168.1.8', 80)

```

IP 地址表示

用 C 编写的网络程序使用数据类型 struct sockaddr 将 IP 地址表示为二进制值 (而不是 Python 程序中常见的地址)。要在 Python 表示和 C 表示之间转换 IPv4 地址, 可以使用 inet_aton() 和 inet_ntoa()。

```

import binascii
import socket
import struct
import sys

for string_address in [ '192.168.1.1', '127.0.0.1' ]:
    packed = socket.inet_aton(string_address)
    print 'Original:', string_address
    print 'Packed   :', binascii.hexlify(packed)
    print 'Unpacked:', socket.inet_ntoa(packed)
    print

```

数据包格式中的 4 个字节可以传递到 C 库、通过网络安全地传输, 或者紧凑地保存在一个数据库。

```
$ python socket_address_packing.py
```

```
Original: 192.168.1.1
Packed  : c0a80101
Unpacked: 192.168.1.1
```

```
Original: 127.0.0.1
Packed  : 7f000001
Unpacked: 127.0.0.1
```

相关函数 `inet_pton()` 和 `inet_ntop()` 都能处理 IPv4 和 IPv6 地址, 根据传入的地址簇参数生成适当的格式。

```
import binascii
import socket
import struct
import sys

string_address = '2002:ac10:10a:1234:21e:52ff:fe74:40e'
packed = socket.inet_pton(socket.AF_INET6, string_address)

print 'Original:', string_address
print 'Packed  :', binascii.hexlify(packed)
print 'Unpacked:', socket.inet_ntop(socket.AF_INET6, packed)
```

IPv6 地址已经是十六进制值, 所以将打包版本转换为一个十六进制数系列时会生成一个与原值类似的串。

```
$ python socket_ipv6_address_packing.py
```

```
Original: 2002:ac10:10a:1234:21e:52ff:fe74:40e
Packed  : 2002ac10010a1234021e52fffe74040e
Unpacked: 2002:ac10:10a:1234:21e:52ff:fe74:40e
```

参见:

IPv6 (<http://en.wikipedia.org/wiki/IPv6>) 维基百科文章, 讨论 Internet Protocol 6 (IPv6)。

OSI Networking Model (http://en.wikipedia.org/wiki/OSI_model) 维基百科文章, 介绍网络实现的 7 层模型。

Assigned Internet Protocol Numbers(www.iana.org/assignments/protocol-numbers/protocol-numbers.xml) 标准协议名和协议码列表。

11.1.2 TCP/IP 客户和服务端

套接字可以配置为一个服务器, 监听到来的消息, 或者也可以配置为客户, 连接到其他应用。TCP/IP 套接字的两端连接后, 可以完成双向通信。

回应服务器

下面这个示例程序以标准库文档中的一个例子为基础, 它接收到来的消息, 再回复给发送

者。首先创建一个 TCP/IP 套接字。

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

然后使用 `bind()` 将这个套接字与服务器地址关联。在这里，地址是 `localhost`（指示当前服务器），端口号为 10000。

```
# Bind the socket to the port
server_address = ('localhost', 10000)
print >>sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)
```

调用 `listen()` 将这个套接字置为服务器模式，调用 `accept()` 等待到来的连接。整数参数是在后台排队的连接数，达到这个连接数后，系统会拒绝连接新客户。这个例子希望一次只处理一个连接。

```
# Listen for incoming connections
sock.listen(1)

while True:
    # Wait for a connection
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()
```

`accept()` 返回服务器和客户之间的一个打开的连接，并返回客户地址。这个连接实际上是另一个端口上的一个不同的套接字（由内核分配）。数据使用 `recv()` 从连接读取，并用 `sendall()` 传输。

```
try:
    print >>sys.stderr, 'connection from', client_address

    # Receive the data in small chunks and retransmit it
    while True:
        data = connection.recv(16)
        print >>sys.stderr, 'received "%s"' % data
        if data:
            print >>sys.stderr, 'sending data back to the client'
            connection.sendall(data)
        else:
            print >>sys.stderr, 'no data from', client_address
            break

finally:
    # Clean up the connection
    connection.close()
```

与一个客户的通信完成时，需要用 `close()` 清理这个连接。这个例子使用了一个 `try:finally` 块来确保 `close()` 总会被调用，即使出现了一个错误也不例外。

回应客户

与服务器不同，客户程序采用另外一种方式建立 `socket`。它不绑定到一个端口并监听，而是使用 `connect()` 将套接字直接关联到远程地址。

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = ('localhost', 10000)
print >>sys.stderr, 'connecting to %s port %s' % server_address
sock.connect(server_address)
```

建立连接之后，可以通过 `socket` 利用 `sendall()` 发送数据，并用 `recv()` 接收数据，这与服务器中是一样的。

```
try:

    # Send data
    message = 'This is the message. It will be repeated.'
    print >>sys.stderr, 'sending "%s"' % message
    sock.sendall(message)

    # Look for the response
    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print >>sys.stderr, 'received "%s"' % data

finally:
    print >>sys.stderr, 'closing socket'
    sock.close()
```

发送了整个消息并接收到一个副本时，套接字会关闭，以释放端口。

客户与服务器

要在不同的终端窗口运行客户和服务端，使它们能够相互通信。服务器输出显示了到来的连接和数据，以及发回给客户的响应。

```
$ python ./socket_echo_server.py
```

```

starting up on localhost port 10000
waiting for a connection
connection from ('127.0.0.1', 52186)
received "This is the mess"
sending data back to the client
received "age. It will be"
sending data back to the client
received " repeated."
sending data back to the client
received ""
no data from ('127.0.0.1', 52186)
waiting for a connection

```

客户输出显示了发出的消息和来自服务器的响应。

```
$ python socket_echo_client.py
```

```

connecting to localhost port 10000
sending "This is the message. It will be repeated."
received "This is the mess"
received "age. It will be"
received " repeated."
closing socket

```

```
$
```

简易客户连接

如果使用便利函数 `create_connection()` 来连接服务器，TCP/IP 客户可以省去几步。这个函数只有一个参数，这是一个包含服务器地址的二值元组，函数将由这个参数推导出用于连接的最佳地址。

```

import socket
import sys

def get_constants(prefix):
    """Create a dictionary mapping socket module
    constants to their names.
    """
    return dict((getattr(socket, n), n)
                 for n in dir(socket)
                 if n.startswith(prefix))

families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')

# Create a TCP/IP socket
sock = socket.create_connection(('localhost', 10000))

```



```

print >>sys.stderr, 'Family :', families[sock.family]
print >>sys.stderr, 'Type   :', types[sock.type]
print >>sys.stderr, 'Protocol:', protocols[sock.proto]
print >>sys.stderr

try:

    # Send data
    message = 'This is the message. It will be repeated.'
    print >>sys.stderr, 'sending "%s"' % message
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print >>sys.stderr, 'received "%s"' % data

    finally:
        print >>sys.stderr, 'closing socket'
        sock.close()

```

`create_connection()` 使用 `getaddrinfo()` 来查找候选连接参数，并返回一个打开的 socket，它的第一个配置可以成功创建一个连接。可以检查 `family`、`type` 和 `proto` 属性确定返回的 socket 类型。

```

$ python socket_echo_client_easy.py
Family   : AF_INET
Type     : SOCK_STREAM
Protocol: IPPROTO_TCP

sending "This is the message. It will be repeated."
received "This is the mess"
received "age. It will be"
received " repeated."
closing socket

```

选择监听地址

将服务器绑定到正确的地址很重要，这样客户才能与之通信。前面的例子都使用 ‘localhost’ 作为 IP 地址，这会限制为只能连接在同一服务器上运行的客户。可以使用服务器的一个公共地址，如 `gethostname()` 返回的值，从而允许其他主机连接。下面这个例子修改了回应服务器，让它监听一个命令行参数指定的地址。

```

import socket
import sys

```



```

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address given on the command line
server_name = sys.argv[1]
server_address = (server_name, 10000)
print >>sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)
sock.listen(1)

while True:
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()
    try:
        print >>sys.stderr, 'client connected:', client_address
        while True:
            data = connection.recv(16)
            print >>sys.stderr, 'received "%s"' % data
            if data:
                connection.sendall(data)
            else:
                break
    finally:
        connection.close()

```

测试这个服务器之前，需要对客户程序做类似的修改。

```

import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port on the server given by the caller
server_address = (sys.argv[1], 10000)
print >>sys.stderr, 'connecting to %s port %s' % server_address
sock.connect(server_address)

try:
    message = 'This is the message. It will be repeated.'
    print >>sys.stderr, 'sending "%s"' % message
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)
    while amount_received < amount_expected:
        data = sock.recv(16)

```

```

        amount_received += len(data)
        print >>sys.stderr, 'received "%s"' % data

    finally:
        sock.close()

```

启动服务器并提供参数 `farnsworth.hellfly.net` 之后, `netstat` 命令显示出它在监听指定的主机地址。

```

$ host farnsworth.hellfly.net

farnsworth.hellfly.net has address 192.168.1.17
$ netstat -an

Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address      Foreign Address    (state)
...
tcp4      0      0 192.168.1.17.10000 *.*                LISTEN
...

```

在另一个主机上运行这个客户时, 传入 `farnsworth.hellfly.net` 作为运行服务器的主机, 会生成以下结果。

```

$ hostname

homer

$ python socket_echo_client_explicit.py farnsworth.hellfly.net

connecting to farnsworth.hellfly.net port 10000
sending "This is the message.  It will be repeated."
received "This is the mess"
received "age.  It will be"
received " repeated."

```

服务器会生成以下输出。

```

$ python ./socket_echo_server_explicit.py farnsworth.hellfly.net

starting up on farnsworth.hellfly.net port 10000
waiting for a connection
client connected: ('192.168.1.8', 57471)
received "This is the mess"
received "age.  It will be"
received " repeated."
received ""
waiting for a connection

```

很多服务器有不止一个网络接口, 相应的会有不止一个 IP 地址。并不需要运行服务的不同副本分别绑定到各个 IP 地址, 可以使用一个特殊的地址 `INADDR_ANY` 同时监听所有地址。

socket 为 INADDR_ANY 定义了一个常量，这是一个整数值，但在传递到 bind() 之前必须将它转换为采用点记法的地址字符串。作为一种快捷方式，可以使用“0.0.0.0”或空串(“”)而不是完成转换。

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address given on the command line
server_address = ('', 10000)
sock.bind(server_address)
print >>sys.stderr, 'starting up on %s port %s' % sock.getsockname()
sock.listen(1)

while True:
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()
    try:
        print >>sys.stderr, 'client connected:', client_address
        while True:
            data = connection.recv(16)
            print >>sys.stderr, 'received "%s"' % data
            if data:
                connection.sendall(data)
            else:
                break
        finally:
            connection.close()
```

要看一个套接字使用的具体地址，可以调用其 getsockname() 方法。启动服务后，再次运行 netstat，可以显示出它在监听所有地址的传入连接。

```
$ netstat -an
```

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
...
tcp4      0      0 *.10000      *.*          LISTEN
...
```

11.1.3 用户数据报客户和服务

用户数据报协议 (user datagram protocol, UDP) 的工作方式与 TCP/IP 不同。TCP 是一个面向流 (stream-oriented) 的协议，确保所有数据以正确的顺序传输，而 UDP 是一个面向消息 (message-oriented) 的协议。UDP 不需要一个长期活动的连接，所以建立 UDP 套接字稍简单一

些。另一方面，UDP 消息必须放在一个数据包中（对于 IPv4，这意味着它们可以包含 65 507 字节，因为 65 535 字节大小的数据包还包括首部信息），而且无法得到 TCP 提供的传输保障。

回应服务器

由于实际上并没有连接，服务器并不需要监听和接受连接。它只需要使用 `bind()` 将其套接字与一个端口关联，然后等待各个消息。

```
import socket
import sys

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind the socket to the port
server_address = ('localhost', 10000)
print >>sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)
```

使用 `recvfrom()` 从套接字读取消息，这个函数会返回数据，还会返回发出这个数据的客户地址。

```
while True:
    print >>sys.stderr, '\nwaiting to receive message'
    data, address = sock.recvfrom(4096)

    print >>sys.stderr, 'received %s bytes from %s' % \
        (len(data), address)
    print >>sys.stderr, data

    if data:
        sent = sock.sendto(data, address)
        print >>sys.stderr, 'sent %s bytes back to %s' % \
            (sent, address)
```

回应客户

UDP 回应客户与服务器类似，但是不使用 `bind()` 将套接字关联到一个地址。它使用 `sendto()` 将消息直接传送到服务器，并使用 `recvfrom()` 接收响应。

```
import socket
import sys

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

server_address = ('localhost', 10000)
message = 'This is the message. It will be repeated.'

try:
```

```

# Send data
print >>sys.stderr, 'sending "%s"' % message
sent = sock.sendto(message, server_address)

# Receive response
print >>sys.stderr, 'waiting to receive'
data, server = sock.recvfrom(4096)
print >>sys.stderr, 'received "%s"' % data

finally:
    print >>sys.stderr, 'closing socket'
    sock.close()

```

客户与服务

运行这个服务器会生成以下输出：

```

$ python ./socket_echo_server_dgram.py

starting up on localhost port 10000

waiting to receive message
received 42 bytes from ('127.0.0.1', 50139)
This is the message.  It will be repeated.
sent 42 bytes back to ('127.0.0.1', 50139)

waiting to receive message

```

客户输出如下：

```

$ python ./socket_echo_client_dgram.py
sending "This is the message.  It will be repeated."
waiting to receive
received "This is the message.  It will be repeated."
closing socket

```

11.1.4 UNIX 域套接字

从程序员的角度来看，使用 UNIX 域套接字和 TCP/IP 套接字存在两个根本区别。首先，套接字的地址是文件系统上的一个路径，而不是一个包含服务器名和端口的元组。其次，文件系统中创建的表示套接字的节点会持久保存，即使套接字关闭也仍然存在，所以每次服务器启动时都需要将其删除。只需在设置部分做一些修改，就可以把前面的回应服务器例子更新为使用 UDS。

```

import socket
import sys
import os

server_address = './uds_socket'

```

```

# Make sure the socket does not already exist
try:
    os.unlink(server_address)
except OSError:
    if os.path.exists(server_address):
        raise

```

需要基于地址簇 AF_UNIX 创建 socket。

```

# Create a UDS socket
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

```

套接字的绑定和到来连接的管理与 TCP/IP 套接字的做法相同。

```

# Bind the socket to the address
print >>sys.stderr, 'starting up on %s' % server_address
sock.bind(server_address)
# Listen for incoming connections
sock.listen(1)

while True:
    # Wait for a connection
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()
    try:
        print >>sys.stderr, 'connection from', client_address

        # Receive the data in small chunks and retransmit it
        while True:
            data = connection.recv(16)
            print >>sys.stderr, 'received "%s"' % data
            if data:
                print >>sys.stderr, 'sending data back to the client'
                connection.sendall(data)
            else:
                print >>sys.stderr, 'no data from', client_address
                break

        finally:
            # Clean up the connection
            connection.close()

```

还需要修改客户设置来使用 UDS。要假设套接字的相应文件系统节点存在，因为服务器要通过绑定这个地址来创建套接字。

```

import socket
import sys

# Create a UDS socket

```

```

sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = './uds_socket'
print >>sys.stderr, 'connecting to %s' % server_address
try:
    sock.connect(server_address)
except socket.error, msg:
    print >>sys.stderr, msg
    sys.exit(1)

```

UDS 客户中发送和接收数据的做法与前面的 TCP/IP 客户是一样的。

```

try:

    # Send data
    message = 'This is the message. It will be repeated.'
    print >>sys.stderr, 'sending "%s"' % message
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print >>sys.stderr, 'received "%s"' % data

finally:
    print >>sys.stderr, 'closing socket'
    sock.close()

```

程序输出基本上相同，但对地址信息有适当的更新。服务器显示接收的消息，并发回给客户。

```
$ python ./socket_echo_server_uds.py
```

```

starting up on ./uds_socket
waiting for a connection
connection from
received "This is the mess"
sending data back to the client
received "age. It will be"
sending data back to the client
received " repeated."
sending data back to the client
received ""
no data from
waiting for a connection

```



客户立即发送所有消息，并采用增量方式逐部分接收消息。

```
$ python socket_echo_client_uds.py

connecting to ./uds_socket
sending "This is the message. It will be repeated."
received "This is the mess"
received "age. It will be"
received " repeated."
closing socket
```

权限

由于 UDS 套接字由文件系统上的一个节点表示，所以可以使用标准文件系统权限来控制对服务器的访问。

```
$ ls -l ./uds_socket

srwxr-xr-x 1 dhellmann dhellmann 0 Sep 20 08:24 ./uds_socket

$ sudo chown root ./uds_socket

$ ls -l ./uds_socket

srwxr-xr-x 1 root dhellmann 0 Sep 20 08:24 ./uds_socket
```

如果客户作为一个用户运行而不是作为 root，现在会导致一个错误，因为进程没有打开套接字的权限。

```
$ python socket_echo_client_uds.py

connecting to ./uds_socket
[Errno 13] Permission denied
```

父进程与子进程间通信

在 UNIX 下，`socketpair()` 函数对于建立 UDS 套接字完成进程间通信很有用。它会创建一对连接的套接字，在创建子进程之后，可以用来在父进程和子进程之间通信。

```
import socket
import os

parent, child = socket.socketpair()

pid = os.fork()

if pid:
    print 'in parent, sending message'
    child.close()
    parent.sendall('ping')
```




```

    response = parent.recv(1024)
    print 'response from child:', response
    parent.close()

else:
    print 'in child, waiting for message'
    parent.close()
    message = child.recv(1024)
    print 'message from parent:', message
    child.sendall('pong')
    child.close()

```

默认地会创建一个 UDS 套接字，不过调用者还可以传递地址簇、套接字类型，甚至协议选项来控制如何创建套接字。

```
$ python socket_socketpair.py
```

```

in child, waiting for message
message from parent: ping
in parent, sending message
response from child: pong

```

11.1.5 组播

点对点连接可以处理很多通信需求，不过随着直接连接数的增加，在多对通信方之间传递相同的消息会变得越来越困难。单独地向各个接收方发送消息会耗费额外的处理时间和带宽，这对于流视频或音频等应用会带来问题。使用组播（multicast）向多个端点同时发送消息则可以得到更好的效率，因为网络基础设施可以确保数据包会传送到所有接收方。

组播消息总是使用 UDP 发送，因为 TCP 需要一个端对端通信通道。组播的地址称为组播组（multicast group），这是常规的 IPv4 地址范围的一个子集（224.0.0.0 到 230.255.255.255），专门为组播通信预留。这些地址会由网络路由器和交换机特殊处理，所以发送到组的消息可以在互联网上分发到加入这个组的所有接收方。

注意：一些托管交换机和路由器默认禁用组播通信。如果运行这个示例程序有问题，可以检查你的网络硬件设置。

发送组播消息

修改后的这个回应客户会向一个组播组发送一个消息，然后报告它收到的所有响应。由于无法知道会收到多少响应，它对套接字使用了一个超时值，以避免等待回答时无限阻塞。

```

import socket
import struct
import sys

```

```
message = 'very important data'
```

```

multicast_group = ('224.3.29.71', 10000)

# Create the datagram socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Set a timeout so the socket does not block indefinitely when trying
# to receive data.
sock.settimeout(0.2)

```

配置这个套接字时还需要提供消息的一个生存时间值 (time-to-live value, TTL)。TTL 会控制多少网络接收这个数据包。使用 `IP_MULTICAST_TTL` 选项和 `setsockopt()` 来设置 TTL。默认值 1 表示路由器不会把数据包转发到当前网段之外。这个值取值范围最大为 255, 应当包装到一个字节中。

```

# Set the time-to-live for messages to 1 so they do not go past the
# local network segment.
ttl = struct.pack('b', 1)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)

```

发送者程序的其余代码类似于 UDP 回应客户, 只是它会接收多个响应, 所以这里使用一个循环来调用 `recvfrom()`, 直到超时。

```

try:

    # Send data to the multicast group
    print >>sys.stderr, 'sending "%s"' % message
    sent = sock.sendto(message, multicast_group)

    # Look for responses from all recipients
    while True:
        print >>sys.stderr, 'waiting to receive'
        try:
            data, server = sock.recvfrom(16)
        except socket.timeout:
            print >>sys.stderr, 'timed out, no more responses'
            break
        else:
            print >>sys.stderr, 'received "%s" from %s' % \
                (data, server)

    finally:
        print >>sys.stderr, 'closing socket'
        sock.close()

```

接收组播消息

建立组播接收者的第一步是创建 UDP 套接字。

```
import socket
```

```

import struct
import sys

multicast_group = '224.3.29.71'
server_address = ('', 10000)

# Create the socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to the server address
sock.bind(server_address)

```

创建常规的套接字并绑定到一个端口后，可以使用 `setsockopt()` 改变 `IP_ADD_MEMBERSHIP` 选项，把它添加到组播组。这个选项值是一个 8 字节的打包表示，包括组播组地址，后面是服务器监听通信流的网络接口（由其 IP 地址标识）。在这里，接收者使用 `INADDR_ANY` 监听所有接口。

```

# Tell the operating system to add the socket to the multicast group
# on all interfaces.
group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

```

接收者的主循环与常规 UDP 回应服务器类似。

```

# Receive/respond loop
while True:
    print >>sys.stderr, '\nwaiting to receive message'
    data, address = sock.recvfrom(1024)

    print >>sys.stderr, 'received %s bytes from %s' % \
        (len(data), address)
    print >>sys.stderr, data

    print >>sys.stderr, 'sending acknowledgement to', address
    sock.sendto('ack', address)

```

示例输出

这个例子显示了组播接收者在两个不同主机上运行。A 地址为 192.168.1.17，B 地址为 192.168.1.8。

```
[A]$ python ./socket_multicast_receiver.py
```

```

waiting to receive message
received 19 bytes from ('192.168.1.17', 51382)
very important data
sending acknowledgement to ('192.168.1.17', 51382)

```

```
[B]$ python ./socket_multicast_receiver.py
```

```
waiting to receive message
received 19 bytes from ('192.168.1.17', 51382)
very important data
sending acknowledgement to ('192.168.1.17', 51382)
```

发送者在主机 A 上运行。

```
$ python ./socket_multicast_sender.py
```

```
sending "very important data"
waiting to receive
received "ack" from ('192.168.1.17', 10000)
waiting to receive
received "ack" from ('192.168.1.8', 10000)
waiting to receive
timed out, no more responses
closing socket
```

消息只发送一次，但是会接收到对发出消息的两个应答，分别来自主机 A 和主机 B。

参见：

Multicast (<http://en.wikipedia.org/wiki/Multicast>) 维基百科文章，介绍组播的技术细节。

IP Multicast (http://en.wikipedia.org/wiki/IP_multicast) 有关 IP 组播的维基百科文章，提供了寻址的有关信息。

11.1.6 发送二进制数据

套接字可以传输字节流。这些字节可能包含文本消息（如前面的例子所示），或者它们也可能由二进制数据构成，这些二进制数据已经编码以便传输。为了完成二进制数据值的传输，要用 struct 把它们打包到一个缓冲区。

下面这个客户程序将一个整数、一个包含两字符的字符串和一个浮点数值编码为一个字节序列，以便传递到套接字完成传输。

```
import binascii
import socket
import struct
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('localhost', 10000)
sock.connect(server_address)

values = (1, 'ab', 2.7)
packer = struct.Struct('I 2s f')
```

```

packed_data = packer.pack(*values)

print 'values =', values

try:

    # Send data
    print >>sys.stderr, 'sending %r' % binascii.hexlify(packed_data)
    sock.sendall(packed_data)

finally:
    print >>sys.stderr, 'closing socket'
    sock.close()

```

在两个系统之间发送多字节的二进制数据时，有一点很重要，要确保连接的两端都知道采用哪一种字节顺序，以及如何把它们重新组装为适合本地体系结构的正确顺序。服务器程序使用了相同的 Struct 指示工具（specifier）来解开接收到的字节，从而以正确的顺序解释。

```

import binascii
import socket
import struct
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('localhost', 10000)
sock.bind(server_address)
sock.listen(1)

unpacker = struct.Struct('I 2s f')

while True:
    print >>sys.stderr, '\nwaiting for a connection'
    connection, client_address = sock.accept()
    try:
        data = connection.recv(unpacker.size)
        print >>sys.stderr, 'received %r' % binascii.hexlify(data)
        unpacked_data = unpacker.unpack(data)
        print >>sys.stderr, 'unpacked:', unpacked_data

    finally:
        connection.close()

```

运行客户会生成以下输出：

```
$ python ./socket_binary_client.py
```

```
values = (1, 'ab', 2.7)
```

```
sending '0100000061620000cdcc2c40'
closing socket
```

服务器显示了它接收的值:

```
$ python ./socket_binary_server.py

waiting for a connection
received '0100000061620000cdcc2c40'
unpacked: (1, 'ab', 2.700000047683716)
```

```
waiting for a connection
```

浮点数值在打包和解包时会损失一些精度, 不过数据确实能按期望的方式传输。有一点要记住, 取决于整数的值, 有时将它转换为文本然后再传输可能更为高效, 而不是使用 `struct`。整数 1 表示为字符串时占用一个字节, 而打包到结构中时会占用 4 个字节。

参见:

`struct` (2.6 节) 在字符串和其他数据类型之间转换。

11.1.7 非阻塞通信和超时

默认情况下, `socket` 配置为发送或接收数据时会阻塞, 在套接字准备就绪之前将停止程序的执行。`send()` 调用等待有缓冲区空间来存放发出的数据, `recv()` 则等待其他程序发出数据以供读取。这种形式的 I/O 操作很容易理解, 不过可能导致操作很低效, 如果两个程序最后都在等待对方发送或接收数据, 甚至会导致死锁。

有很多种方法来绕开这种情况。一种做法是对各个套接字分别使用单独的线程完成通信。不过, 这可能引入线程间通信的其他复杂性。另一种选择是将套接字改为根本不阻塞, 即使没有准备好来处理操作, 也会立即返回。可以使用 `setblocking()` 方法改变一个套接字的阻塞标志。默认值为 1, 这表示会阻塞。传入值 0 则会关闭阻塞。如果套接字将阻塞关闭, 而且没有为处理操作做好准备, 则会产生一个 `socket.error`。

一种折衷的解决方案是为套接字操作设置一个超时值。可以使用 `settimeout()` 将一个 `socket` 的超时值改为一个浮点数值, 表示在确定这个套接字未做好操作准备之前阻塞的时间 (秒数)。超过这个超时期限时, 会产生一个 `timeout` 异常。

参见:

`socket` (<http://docs.python.org/library/socket.html>) 这个模块的标准库文档。

Socket Programming HOWTO (<http://docs.python.org/howto/sockets.html>) Gordon McMillan 提供的一个介绍性指南, 包含在标准库文档中。

`select` (11.2 节) 测试一个套接字, 查看是否准备好完成非阻塞 I/O 读写。

`SocketServer` (11.3 节) 创建网络服务器的框架。

`urllib` (12.3 节) 和 `urllib2` (12.4.7 节) 大多数网络客户都应当使用更方便的库通过 URL 来访问远程资源。

asyncore (11.4 节) 和 asynchat (11.5 节) 异步通信的框架。

Unix Network Programming, Volume 1: The Sockets Networking API, 3/E, W. Richard Stevens、Bill Fenner 和 Andrew M. Rudoff。由 Addison-Wesley Professional 出版, 2004. ISBN-10: 0131411551

11.2 select——高效等待 I/O

作用: 等待输入或输出通道已经准备就绪的通知。

Python 版本: 1.4 及以后版本

select 模块允许访问特定于平台的 I/O 监视函数。最可移植的接口是 POSIX 函数 select(); UNIX 和 Windows 都提供了这个函数。这个模块还包括函数 poll() (这个 API 只适用于 UNIX), 另外还提供了很多只适用于一些 UNIX 特定变种的选项。

11.2.1 使用 select()

Python 的 select() 函数是底层操作系统实现的一个直接接口。它会监视套接字、打开的文件和管道 (可以是任何有 fileno() 方法的对象, 这个方法会返回一个合法的文件描述符), 直到这个对象可读或可写, 或者出现一个通信错误。利用 select() 可以更为容易地同时监视多个连接, 这比在 Python 中使用套接字超时编写一个轮询循环更为高效, 因为监视发生在操作系统网络层而不是在解释器中完成。

注意: 对 Python 文件对象使用 select() 只适用于 UNIX, 在 Windows 下并不支持。

可以扩展 socket 一节的回应服务器例子, 通过使用 select() 来同时监视多个连接。这个新版本首先创建一个非阻塞的 TCP/IP 套接字, 将它配置为监听一个地址。

```
import select
import socket
import sys
import Queue

# Create a TCP/IP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

# Bind the socket to the port
server_address = ('localhost', 10000)
print '>>sys.stderr, 'starting up on %s port %s' % server_address
server.bind(server_address)

# Listen for incoming connections
server.listen(5)
```

`select()` 的参数是 3 个列表，包含要监视的通信通道。将检查第一个对象列表中的对象来得到要读取的数据，第二个列表中包含的对象将接收发出的数据（如果缓冲区有空间），第三个列表包含那些可能有错误的对象（通常是输入和输出通道对象的组合）。在服务器中，下一步是建立这些列表，其中包含要传至 `select()` 的输入源和输出目标。

```
# Sockets from which we expect to read
inputs = [ server ]
```

```
# Sockets to which we expect to write
outputs = [ ]
```

由服务器主循环向这些列表添加或删除连接。由于这个版本的服务器在发送数据之前要等待一个套接字变为可写（而不是立即发送应答），所以每个输出连接都需要一个队列，作为通过这个套接字发送的数据的缓冲区。

```
# Outgoing message queues (socket:Queue)
message_queues = {}
```

服务器程序的主要部分是一个循环，调用 `select()` 来阻塞，并等待网络活动。

```
while inputs:
```

```
    # Wait for at least one of the sockets to be ready for processing
    print >>sys.stderr, 'waiting for the next event'
    readable, writable, exceptional = select.select(inputs,
                                                    outputs,
                                                    inputs)
```

`select()` 返回 3 个新列表，包含所传入列表内容的子集。`readable` 列表中的所有套接字会缓存到来的数据，可供读取。`writable` 列表中所有套接字的缓冲区中有自由空间，可以写入数据。`exceptional` 中返回的套接字都有一个错误（“异常条件”的具体定义取决于平台）。

“可读”套接字表示 3 种可能的情况。如果套接字是主“服务器”套接字，即用来监听连接的套接字，“可读”条件则意味着它已经准备就绪，可以接受另一个到来的连接。除了将新连接添加到要监视的输入列表之外，这一部分还将客户套接字设置为非阻塞。

```
# Handle inputs
for s in readable:
    if s is server:
        # A "readable" socket is ready to accept a connection
        connection, client_address = s.accept()
        print >>sys.stderr, ' connection from', client_address
        connection.setblocking(0)
        inputs.append(connection)

    # Give the connection a queue for data we want to send
    message_queues[connection] = Queue.Queue()
```


下一种情况是一个已建立的连接，客户已经发送了数据。数据用 `recv()` 读取，然后放置在队列中，以便通过套接字发送并返回给客户。

```
else:
    data = s.recv(1024)
    if data:
        # A readable client socket has data
        print >>sys.stderr, ' received "%s" from %s' % \
            (data, s.getpeername())
        message_queues[s].put(data)
        # Add output channel for response
        if s not in outputs:
            outputs.append(s)
```

没有可用数据的可读套接字来自已经断开连接的客户，此时可以关闭流。

```
else:
    # Interpret empty result as closed connection
    print >>sys.stderr, ' closing', client_address
    # Stop listening for input on the connection
    if s in outputs:
        outputs.remove(s)
    inputs.remove(s)
    s.close()

    # Remove message queue
    del message_queues[s]
```

对于可写连接，情况要少一些。如果对应一个连接的队列中有数据，则发送下一个消息。否则，将这个连接从输出连接列表中删除，下一次循环时，`select()` 不再指示这个套接字已准备就绪可以发送数据。

```
# Handle outputs
for s in writable:
    try:
        next_msg = message_queues[s].get_nowait()
    except Queue.Empty:
        # No messages waiting so stop checking for writability.
        print >>sys.stderr, ' ', s.getpeername(), 'queue empty'
        outputs.remove(s)
    else:
        print >>sys.stderr, ' sending "%s" to %s' % \
            (next_msg, s.getpeername())
        s.send(next_msg)
```

最后一点，如果一个套接字有错误，则会关闭。

```
# Handle "exceptional conditions"
for s in exceptional:
    print >>sys.stderr, 'exception condition on', s.getpeername()
```

```

# Stop listening for input on the connection
inputs.remove(s)
if s in outputs:
    outputs.remove(s)
s.close()

# Remove message queue
del message_queues[s]

```

这个示例客户程序使用了两个套接字来展示服务器如何利用 `select()` 同时管理多个连接。客户首先将各个 TCP/IP 套接字连接到服务器。

```

import socket
import sys

messages = [ 'This is the message. ',
              'It will be sent ',
              'in parts.',
              ]

server_address = ('localhost', 10000)
# Create a TCP/IP socket
socks = [ socket.socket(socket.AF_INET, socket.SOCK_STREAM),
          socket.socket(socket.AF_INET, socket.SOCK_STREAM),
          ]

# Connect the socket to the port where the server is listening
print >>sys.stderr, 'connecting to %s port %s' % server_address
for s in socks:
    s.connect(server_address)

```

然后通过各个套接字一次发送一段消息，写新数据之后再读取得到的所有响应。

```

for message in messages:

    # Send messages on both sockets
    for s in socks:
        print >>sys.stderr, '%s: sending "%s"' % \
            (s.getsockname(), message)
        s.send(message)

    # Read responses on both sockets
    for s in socks:
        data = s.recv(1024)
        print >>sys.stderr, '%s: received "%s"' % \
            (s.getsockname(), data)
        if not data:
            print >>sys.stderr, 'closing socket', s.getsockname()
            s.close()

```

在一个窗口中运行服务器，在另一个窗口中运行客户程序。输入如下所示（端口号有所不同）。

```
$ python ./select_echo_server.py

starting up on localhost port 10000
waiting for the next event
  connection from ('127.0.0.1', 55472)
waiting for the next event
  connection from ('127.0.0.1', 55473)
  received "This is the message. " from ('127.0.0.1', 55472)
waiting for the next event
  received "This is the message. " from ('127.0.0.1', 55473)
  sending "This is the message. " to ('127.0.0.1', 55472)
waiting for the next event
  ('127.0.0.1', 55472) queue empty
  sending "This is the message. " to ('127.0.0.1', 55473)
waiting for the next event
  ('127.0.0.1', 55473) queue empty
waiting for the next event
  received "It will be sent " from ('127.0.0.1', 55472)
  received "It will be sent " from ('127.0.0.1', 55473)
waiting for the next event
  sending "It will be sent " to ('127.0.0.1', 55472)
  sending "It will be sent " to ('127.0.0.1', 55473)
waiting for the next event
  ('127.0.0.1', 55472) queue empty
  ('127.0.0.1', 55473) queue empty
waiting for the next event
  received "in parts." from ('127.0.0.1', 55472)
  received "in parts." from ('127.0.0.1', 55473)
waiting for the next event
  sending "in parts." to ('127.0.0.1', 55472)
  sending "in parts." to ('127.0.0.1', 55473)
waiting for the next event
  ('127.0.0.1', 55472) queue empty
  ('127.0.0.1', 55473) queue empty
waiting for the next event
  closing ('127.0.0.1', 55473)
  closing ('127.0.0.1', 55473)
waiting for the next event
```

客户程序的输出显示了使用这两个套接字发送和接收到的数据。

```
$ python ./select_echo_multiclient.py

connecting to localhost port 10000
('127.0.0.1', 55821): sending "This is the message. "
('127.0.0.1', 55822): sending "This is the message. "
```

```

('127.0.0.1', 55821): received "This is the message. "
('127.0.0.1', 55822): received "This is the message. "
('127.0.0.1', 55821): sending "It will be sent "
('127.0.0.1', 55822): sending "It will be sent "
('127.0.0.1', 55821): received "It will be sent "
('127.0.0.1', 55822): received "It will be sent "
('127.0.0.1', 55821): sending "in parts."
('127.0.0.1', 55822): sending "in parts."
('127.0.0.1', 55821): received "in parts."
('127.0.0.1', 55822): received "in parts."

```

11.2.2 有超时的非阻塞 I/O

`select()` 还可以有第 4 个参数 (可选), 如果没有活动通道, 它在停止监视之前会等待一定时间, 这个参数就是在此之前等待的秒数。通过使用一个超时值, 可以让主程序在一个更大的处理循环中调用 `select()`, 在检查网络输入的间隙接受其他动作。

超过这个超时期限时, `select()` 会返回 3 个空列表。更新前面的服务器示例, 使用一个超时值, 这需要向 `select()` 调用添加一个额外的参数, 并在 `select()` 返回后处理空列表。

```

# Wait for at least one of the sockets to be ready for processing
print >>sys.stderr, '\nwaiting for the next event'
timeout = 1
readable, writable, exceptional = select.select(inputs,
                                                outputs,
                                                inputs,
                                                timeout)

if not (readable or writable or exceptional):
    print >>sys.stderr, ' timed out, do some other work here'
    continue

```

客户程序的这个“慢”版本会在发送各个消息之后暂停, 模拟传输中的时延或其他延迟。

```

import socket
import sys
import time

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = ('localhost', 10000)
print >>sys.stderr, 'connecting to %s port %s' % server_address
sock.connect(server_address)
time.sleep(1)

messages = [ 'Part one of the message.',
             'Part two of the message.',

```

```

    ]
    amount_expected = len(''.join(messages))

    try:

        # Send data
        for message in messages:
            print >>sys.stderr, 'sending "%s"' % message
            sock.sendall(message)
            time.sleep(1.5)

        # Look for the response
        amount_received = 0

        while amount_received < amount_expected:
            data = sock.recv(16)
            amount_received += len(data)
            print >>sys.stderr, 'received "%s"' % data

    finally:
        print >>sys.stderr, 'closing socket'
        sock.close()

```

运行这个新服务器和慢客户，服务器会生成以下结果：

```

$ python ./select_echo_server_timeout.py

starting up on localhost port 10000
waiting for the next event
  connection from ('127.0.0.1', 55480)
waiting for the next event
  received "Part one of the message." from ('127.0.0.1', 55480)
waiting for the next event
  sending "Part one of the message." to ('127.0.0.1', 55480)
waiting for the next event
  ('127.0.0.1', 55480) queue empty
waiting for the next event
  received "Part two of the message." from ('127.0.0.1', 55480)
waiting for the next event
  sending "Part two of the message." to ('127.0.0.1', 55480)
waiting for the next event
  ('127.0.0.1', 55480) queue empty
waiting for the next event
  closing ('127.0.0.1', 55480)
waiting for the next event

```

以下是客户输出：

```

$ python ./select_echo_slow_client.py

```

```

connecting to localhost port 10000
sending "Part one of the message."
sending "Part two of the message."
received "Part one of the "
received "message.Part two"
received " of the message."
closing socket

```

11.2.3 使用 poll()

poll() 函数提供的特性与 select() 类似, 不过底层实现更为高效。其缺点是 poll() 在 Windows 下不支持, 所以使用 poll() 的程序可移植性较差。

利用 poll() 建立的回应服务器最前面与其他例子一样, 也使用了相同的套接字配置代码。

```

import select
import socket
import sys
import Queue

# Create a TCP/IP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

# Bind the socket to the port
server_address = ('localhost', 10000)
print '>>sys.stderr, 'starting up on %s port %s' % server_address
server.bind(server_address)

# Listen for incoming connections
server.listen(5)
# Keep up with the queues of outgoing messages
message_queues = {}

```

传入 poll() 的超时值用毫秒表示, 而不是秒, 所以如果要暂停 1 秒, 超时值必须设置为 1000。

```

# Do not block forever (milliseconds)
TIMEOUT = 1000

```

Python 用一个类来实现 poll(), 由这个类管理所监视的注册数据通道。通道通过调用 register() 添加, 同时利用标志指示该通道关注哪些事件。表 11.1 列出了所有标志。

表 11.1 poll() 的事件标志

事 件	描 述
POLLIN	输入准备就绪
POLLPRI	优先级输入准备就绪

(续)

事 件	描 述
POLLOUT	能够接收输出
POLLERR	错误
POLLHUP	通道关闭
POLLNVAL	通道未打开

回应服务器将建立一些只用于读的套接字，另外一些套接字可以读或写。局部变量 `READ_ONLY` 和 `READ_WRITE` 中保存了相应的标志组合。

```
# Commonly used flag sets
READ_ONLY = ( select.POLLIN |
               select.POLLPRI |
               select.POLLHUP |
               select.POLLERR )
READ_WRITE = READ_ONLY | select.POLLOUT
```

这里注册了 `server` 套接字，到来的连接或数据会触发一个事件。

```
# Set up the poller
poller = select.poll()
poller.register(server, READ_ONLY)
```

由于 `poll()` 返回一个元组列表，元组中包含套接字的文件描述符和事件标志，因此需要一个文件描述符编号到对象的映射来获取 `socket`，以便读取或写入。

```
# Map file descriptors to socket objects
fd_to_socket = { server.fileno(): server,
                 }
```

服务器的循环调用 `poll()`，然后处理通过查找套接字返回的“事件”，并根据事件中的标志采取行动。

```
while True:

    # Wait for at least one of the sockets to be ready for processing
    print >>sys.stderr, 'waiting for the next event'
    events = poller.poll(TIMEOUT)

    for fd, flag in events:

        # Retrieve the actual socket from its file descriptor
        s = fd_to_socket[fd]
```

与 `select()` 类似，如果主服务器套接字是“可读的”，实际上这表示有一个来自客户的连接。这个新连接用 `READ_ONLY` 标志注册，用来监视通过它的新数据。

```
# Handle inputs
if flag & (select.POLLIN | select.POLLPRI):
```

```

if s is server:
    # A readable socket is ready to accept a connection
    connection, client_address = s.accept()
    print >>sys.stderr, ' connection', client_address
    connection.setblocking(0)
    fd_to_socket[ connection.fileno() ] = connection
    poller.register(connection, READ_ONLY)

    # Give the connection a queue for data to send
    message_queues[connection] = Queue.Queue()

```

除了服务器以外，其他套接字都是现有的客户，其数据已经缓存，并等待读取。可以使用 `recv()` 从缓冲区获取数据。

```

else:
    data = s.recv(1024)

```

如果 `recv()` 返回了数据，会放置在这个套接字的相应发出队列中，套接字的标志使用 `modify()` 改变，使 `poll()` 监视这个套接字是否准备好接收数据。

```

if data:
    # A readable client socket has data
    print >>sys.stderr, ' received "%s" from %s' % \
        (data, s.getpeername())
    message_queues[s].put(data)
    # Add output channel for response
    poller.modify(s, READ_WRITE)

```

`recv()` 返回的空串表示客户已经断开连接，所以使用 `unregister()` 告诉 `poll` 对象忽略这个套接字。

```

else:
    # Interpret empty result as closed connection
    print >>sys.stderr, ' closing', client_address
    # Stop listening for input on the connection
    poller.unregister(s)
    s.close()

    # Remove message queue
    del message_queues[s]

```

`POLLHUP` 标志指示一个客户“挂起”连接而没有将其妥善地关闭。服务器不会轮询消失的客户。

```

elif flag & select.POLLHUP:
    # Client hung up
    print >>sys.stderr, ' closing', client_address, '(HUP)'
    # Stop listening for input on the connection
    poller.unregister(s)
    s.close()

```


可写套接字的处理看起来与 select() 例子中的版本类似, 不过使用了 modify() 来改变轮询服务器中套接字的标志, 而不是将其从输出列表中删除。

```
elif flag & select.POLLOUT:
    # Socket is ready to send data, if there is any to send.
    try:
        next_msg = message_queues[s].get_nowait()
    except Queue.Empty:
        # No messages waiting so stop checking
        print >>sys.stderr, s.getpeername(), 'queue empty'
        poller.modify(s, READ_ONLY)
    else:
        print >>sys.stderr, ' sending "%s" to %s' % \
            (next_msg, s.getpeername())
        s.send(next_msg)
```

最后一点, 任何有 POLLERR 标志的事件会导致服务器关闭套接字。

```
elif flag & select.POLLERR:
    print >>sys.stderr, ' exception on', s.getpeername()
    # Stop listening for input on the connection
    poller.unregister(s)
    s.close()

    # Remove message queue
    del message_queues[s]
```

基于轮询的服务器与 select_echo_multiclient.py(使用多个套接字的客户程序)一起运行时, 输出如下:

```
$ python ./select_poll_echo_server.py

waiting for the next event
waiting for the next event
connection ('127.0.0.1', 62835)
waiting for the next event
connection ('127.0.0.1', 62836)
waiting for the next event
received "This is the message. " from ('127.0.0.1', 62835)
waiting for the next event
sending "This is the message. " to ('127.0.0.1', 62835)
waiting for the next event
('127.0.0.1', 62835) queue empty
waiting for the next event
received "This is the message. " from ('127.0.0.1', 62836)
waiting for the next event
sending "This is the message. " to ('127.0.0.1', 62836)
waiting for the next event
('127.0.0.1', 62836) queue empty
```

```
waiting for the next event
  received "It will be sent " from ('127.0.0.1', 62835)
waiting for the next event
  sending "It will be sent " to ('127.0.0.1', 62835)
waiting for the next event
('127.0.0.1', 62835) queue empty
waiting for the next event
  received "It will be sent " from ('127.0.0.1', 62836)
waiting for the next event
  sending "It will be sent " to ('127.0.0.1', 62836)
waiting for the next event
('127.0.0.1', 62836) queue empty
waiting for the next event
  received "in parts." from ('127.0.0.1', 62835)
  received "in parts." from ('127.0.0.1', 62836)
waiting for the next event
  sending "in parts." to ('127.0.0.1', 62835)
  sending "in parts." to ('127.0.0.1', 62836)
waiting for the next event
('127.0.0.1', 62835) queue empty
('127.0.0.1', 62836) queue empty
waiting for the next event
  closing ('127.0.0.1', 62836)
  closing ('127.0.0.1', 62836)
waiting for the next event
```

11.2.4 平台特定选项

select 还提供了一些可移植性较差的选项，包括 `epoll`（Linux 支持的边界轮询 API）、`kqueue`（使用了 BSD 的内核队列）和 `kevent`（BSD 的内核事件接口）。有关这些选项如何工作，更多详细内容请参考操作系统库文档。

参见：

`select` (<http://docs.python.org/library/select.html>) 这个模块的标准库文档。

`Socket Programming HOWTO` (<http://docs.python.org/howto/sockets.html>) Gordon McMillan 撰写的一个指导性指南，收录在标准库文档中。

`socket` (11.1 节) 底层网络通信。

`SocketServer` (11.3 节) 创建服务器应用的框架。

`asyncore` (11.4 节) 和 `asynchat` (11.5 节) 异步 I/O 框架。

UNIX Network Programming, Volume 1: The Sockets Networking API, 3/E W. Richard Stevens、Bill Fenner 和 Andrew M. Rudoff。由 Addison-Wesley Professional 2004 年出版。ISBN-10: 0131411551。

11.3 SocketServer——创建网络服务器

作用：创建网络服务器。

Python 版本：1.4 及以后版本

SocketServer 模块是创建网络服务器的一个框架。它定义了一些类来处理 TCP、UDP、UNIX 流和 UNIX 数据报之上的同步网络请求（服务器请求处理器会阻塞，直至请求完成）。它还提供了一些“混入”类^①（mix-in），可以很容易地转换服务器，为每个请求使用一个单独的线程或进程。

处理请求的责任划分到一个服务器类和一个请求处理器类。服务器处理通信问题，如监听一个套接字并接受连接，请求处理器处理“协议”问题，如解释到来的数据、处理数据并把数据发回给客户。这种责任划分意味着很多应用可以使用某个现有的服务器类，而不需要任何修改，另外可以提供一个请求与各个其他处理器类通信，由这些处理器类处理定制协议。

11.3.1 服务器类型

SocketServer 中定义了 5 个不同的服务器类。BaseServer 定义了 API，这个类不应实例化，也不能直接使用。TCPServer 使用 TCP/IP 套接字通信，UDPServer 使用数据报套接字。UnixStreamServer 和 UnixDatagramServer 使用 UNIX 域套接字，只适用于 UNIX 平台。

11.3.2 服务器对象

要构造一个服务器，需要向它传递一个地址（服务器将在这个地址监听请求），以及一个请求处理器类（而非实例）。地址格式取决于所使用的服务器类型和套接字簇。可以参考 socket 模块文档来了解有关的详细内容。

一旦实例化服务器对象，可以使用 handle_request() 或 serve_forever() 处理请求。serve_forever() 方法在一个无限循环中调用 handle_request()，不过如果应用需要将服务器与另一个事件循环集成，或者使用 select() 来监视对应不同服务器的多个套接字，也可以直接调用 handle_request()。

11.3.3 实现服务器

创建一个服务器时，通常重用某个现有的类并提供一个定制请求处理器类就足够了。但针对另外一些情况，BaseServer 还包含了一些可以在子类中覆盖的方法。

- verify_request(request, client_address)：返回 True 表示要处理请求，返回 False 则忽略请求。例如，一个服务器可能拒绝来自一个 IP 段的请求，或者因为服务器负载过重而拒绝请求。
- process_request(request, client_address)：调用 finish_request() 具体完成处理请求的工作。

① mix-in 的作用是，在运行期间动态改变类的基类或类的方法，使类的表现可以发生变化。——译者注

它还可以创建一个单独的线程或进程，就像 `mix-in` 类的做法一样。

- `finish_request(request, client_address)`：使用提供给服务器构造函数的类创建一个请求处理器实例。在这个请求处理器上调用 `handle()` 来处理请求。

11.3.4 请求处理器

要接收到来的请求以及确定采取什么行为，其中大部分工作都由请求处理器完成。处理器负责在套接字层之上实现协议（即 HTTP、XML-RPC 或 AMQP）。请求处理器从到来数据通道读取请求，处理这个请求，并写回一个响应。要覆盖 3 个方法。

- `setup()`：为请求准备请求处理器。在 `StreamRequestHandler` 中，`setup()` 方法会创建类文件的对象来读写套接字。
- `handle()`：对请求完成具体工作。解析到来的请求，处理数据，并发出一个响应。
- `finish()`：清理 `setup()` 期间创建的所有数据。

很多处理器实现时可以只有一个 `handle()` 方法。

11.3.5 回应示例

下面这个例子实现了一对简单的服务器 / 请求处理器，将接受 TCP 连接，并回应客户发送的所有数据。首先来看请求处理器。

```
import logging
import sys
import SocketServer
logging.basicConfig(level=logging.DEBUG,
                    format='%(name)s: %(message)s',
                    )

class EchoRequestHandler(SocketServer.BaseRequestHandler):

    def __init__(self, request, client_address, server):
        self.logger = logging.getLogger('EchoRequestHandler')
        self.logger.debug('__init__')
        SocketServer.BaseRequestHandler.__init__(self, request,
                                                  client_address,
                                                  server)

    def setup(self):
        self.logger.debug('setup')
        return SocketServer.BaseRequestHandler.setup(self)

    def handle(self):
        self.logger.debug('handle')
```

```

    # Echo the back to the client
    data = self.request.recv(1024)
    self.logger.debug('recv()->"%s"', data)
    self.request.send(data)
    return

def finish(self):
    self.logger.debug('finish')
    return SocketServer.BaseRequestHandler.finish(self)

```

真正需要实现的只有一个方法，即 `EchoRequestHandler.handle()`，不过这里包含了前面提到的所有方法（`setup()`、`handle()` 和 `finish()`），以展示调用顺序。`EchoServer` 类的工作与 `TCPServer` 相同，只不过在调用各个方法时会记录日志。

```

class EchoServer(SocketServer.TCPServer):

    def __init__(self, server_address,
                 handler_class=EchoRequestHandler,
                 ):
        self.logger = logging.getLogger('EchoServer')
        self.logger.debug('__init__')
        SocketServer.TCPServer.__init__(self, server_address,
                                         handler_class)

    def server_activate(self):
        self.logger.debug('server_activate')
        SocketServer.TCPServer.server_activate(self)
        return

    def serve_forever(self, poll_interval=0.5):
        self.logger.debug('waiting for request')
        self.logger.info('Handling requests, press <Ctrl-C> to quit')
        SocketServer.TCPServer.serve_forever(self, poll_interval)
        return

    def handle_request(self):
        self.logger.debug('handle_request')
        return SocketServer.TCPServer.handle_request(self)

    def verify_request(self, request, client_address):
        self.logger.debug('verify_request(%s, %s)',
                          request, client_address)
        return SocketServer.TCPServer.verify_request(self, request,
                                                    client_address)

    def process_request(self, request, client_address):

```

```

        self.logger.debug('process_request(%s, %s)',
                           request, client_address)
        return SocketServer.TCPServer.process_request(self, request,
                                                       client_address)

    def server_close(self):
        self.logger.debug('server_close')
        return SocketServer.TCPServer.server_close(self)

    def finish_request(self, request, client_address):
        self.logger.debug('finish_request(%s, %s)',
                           request, client_address)
        return SocketServer.TCPServer.finish_request(self, request,
                                                       client_address)

    def close_request(self, request_address):
        self.logger.debug('close_request(%s)', request_address)
        return SocketServer.TCPServer.close_request(self,
                                                       request_address)

    def shutdown(self):
        self.logger.debug('shutdown()')
        return SocketServer.TCPServer.shutdown(self)

```

最后一步是增加一个主程序，它会建立服务器，使之在一个线程中运行，并向这个服务器发送数据，来展示回应数据时会调用哪些方法。

```

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel assign a port
    server = EchoServer(address, EchoRequestHandler)
    ip, port = server.server_address # what port was assigned?

    # Start the server in a thread
    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    logger = logging.getLogger('client')
    logger.info('Server on %s:%s', ip, port)

    # Connect to the server
    logger.debug('creating socket')
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logger.debug('connecting to server')
    s.connect((ip, port))

```

```

# Send the data
message = 'Hello, world'
logger.debug('sending data: "%s"', message)
len_sent = s.send(message)

# Receive a response
logger.debug('waiting for response')
response = s.recv(len_sent)
logger.debug('response from server: "%s"', response)

# Clean up
server.shutdown()
logger.debug('closing socket')
s.close()
logger.debug('done')
server.socket.close()

```

运行这个程序会生成以下输出。

```
$ python SocketServer_echo.py
```

```

EchoServer: __init__
EchoServer: server_activate
EchoServer: waiting for request
EchoServer: Handling requests, press <Ctrl-C> to quit
client: Server on 127.0.0.1:62859
client: creating socket
client: connecting to server
EchoServer: verify_request(<socket._socketobject object at 0x100e1b8
a0>, ('127.0.0.1', 62860))
EchoServer: process_request(<socket._socketobject object at 0x100e1b
8a0>, ('127.0.0.1', 62860))
EchoServer: finish_request(<socket._socketobject object at 0x100e1b8
a0>, ('127.0.0.1', 62860))
EchoRequestHandler: __init__
EchoRequestHandler: setup
EchoRequestHandler: handle
client: sending data: "Hello, world"
EchoRequestHandler: recv()->"Hello, world"
EchoRequestHandler: finish
EchoServer: close_request(<socket._socketobject object at 0x100e1b8a
0>)
client: waiting for response
client: response from server: "Hello, world"
EchoServer: shutdown()
client: closing socket
client: done

```

注意：每次程序运行时使用的端口号会改变，因为内核会自动分配可用的端口。要让服务器每次监听一个特定的端口，需要在地址元组中提供该端口号而不是 0。

以下是这个服务器的“压缩”版本，这里没有日志记录调用。请求处理器类中只需要提供 `handle()` 方法。

```
import SocketServer

class EchoRequestHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        self.request.send(data)
        return

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel assign a port
    server = SocketServer.TCPServer(address, EchoRequestHandler)
    ip, port = server.server_address # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'
    print 'Sending: "%s"' % message
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(len_sent)
    print 'Received: "%s"' % response

    # Clean up
    server.shutdown()
    s.close()
    server.socket.close()
```

这里并不需要特殊的服务器类，因为 `TCPServer` 会处理所有服务器需求。


```
$ python SocketServer_echo_simple.py
```

```
Sending : "Hello, world"
```

```
Received: "Hello, world"
```

11.3.6 线程和进程

要为一个服务器增加线程或进程 (forking) 支持, 需要在服务器的类层次结构中包括适当的 mix-in 类。这些 mix-in 类要覆盖 `process_request()`, 从而当准备处理一个请求时开始一个新的线程或进程, 具体工作则在这个新的子线程或进程中完成。

对于线程, 要使用 `ThreadingMixIn`。

```
import threading
import SocketServer

class ThreadedEchoRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_thread = threading.currentThread()
        response = '%s: %s' % (cur_thread.getName(), data)
        self.request.send(response)
        return

class ThreadedEchoServer(SocketServer.ThreadingMixIn,
                        SocketServer.TCPServer,
                        ):

    pass

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel assign a port
    server = ThreadedEchoServer(address, ThreadedEchoRequestHandler)
    ip, port = server.server_address # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True) # don't hang on exit
    t.start()
    print 'Server loop running in thread:', t.getName()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))
```

```

# Send the data
message = 'Hello, world'
print 'Sending : "%s"' % message
len_sent = s.send(message)

# Receive a response
response = s.recv(1024)
print 'Received: "%s"' % response

# Clean up
server.shutdown()
s.close()
server.socket.close()

```

对于这个使用线程的服务器，其响应包含了处理请求的线程的标识符。

```
$ python SocketServer_threaded.py
```

```

Server loop running in thread: Thread-1
Sending : "Hello, world"
Received: "Thread-2: Hello, world"

```

对于不同的进程，要使用 `ForkingMixIn`。

```

import os
import SocketServer

class ForkingEchoRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_pid = os.getpid()
        response = '%s: %s' % (cur_pid, data)
        self.request.send(response)
        return

class ForkingEchoServer(SocketServer.ForkingMixIn,
                        SocketServer.TCPServer,
                        ):

    pass

if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0) # let the kernel assign a port
    server = ForkingEchoServer(address, ForkingEchoRequestHandler)
    ip, port = server.server_address # what port was assigned?

```

```

t = threading.Thread(target=server.serve_forever)
t.setDaemon(True) # don't hang on exit
t.start()
print 'Server loop running in process:', os.getpid()

# Connect to the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, port))

# Send the data
message = 'Hello, world'
print 'Sending : "%s"' % message
len_sent = s.send(message)

# Receive a response
response = s.recv(1024)
print 'Received: "%s"' % response

# Clean up
server.shutdown()
s.close()
server.socket.close()

```

在这里，服务器的响应中包含了子进程的进程 id。

```
$ python SocketServer_forking.py
```

```

Server loop running in process: 12797
Sending : "Hello, world"
Received: "12798: Hello, world"

```

参见：

SocketServer (<http://docs.python.org/lib/module-SocketServer.html>) 这个模块的标准库文档。

asyncore (11.4 节) 使用 asyncore 创建异步服务器，它在处理请求时不会阻塞。

SimpleXMLRPCServer (12.11 节) 使用 SocketServer 构建的 XML-RPC 服务器。

11.4 asyncore——异步 I/O

作用：异步 I/O 处理器。

Python 版本：1.5.2 及以后版本

asyncore 模块包括一些工具来处理 I/O 对象，如套接字，从而能异步地管理这些对象（而不是使用多个线程或进程）。其中包括的主要的类是 dispatcher，这是套接字的一个包装器，提供了一些 hook（钩子），从主循环函数 loop() 调用时可以处理连接以及读写事件。

11.4.1 服务器

下面这个例子展示了如何在服务器和客户中使用 `asyncore`，这里重新实现了 `SocketServer` 示例中的 `EchoServer`。这个新实现中使用了 3 个类。第一个类是 `EchoServer`，它从客户接收到的连接。一旦接受第一个连接，这个演示实现就关闭，这样一来，尝试运行代码时可以更容易地开始和停止服务器。

```
import asyncore
import logging

class EchoServer(asyncore.dispatcher):
    """Receives connections and establishes handlers for each client.
    """

    def __init__(self, address):
        self.logger = logging.getLogger('EchoServer')
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(address)
        self.address = self.socket.getsockname()
        self.logger.debug('binding to %s', self.address)
        self.listen(1)
        return

    def handle_accept(self):
        # Called when a client connects to the socket
        client_info = self.accept()
        self.logger.debug('handle_accept() -> %s', client_info[1])
        EchoHandler(sock=client_info[0])
        # Only deal with one client at a time,
        # so close as soon as the handler is set up.
        # Under normal conditions, the server
        # would run forever or until it received
        # instructions to stop.
        self.handle_close()
        return

    def handle_close(self):
        self.logger.debug('handle_close()')
        self.close()
        return
```

`handle_accept()` 中每次接受一个新连接时，`EchoServer` 会创建一个新的 `EchoHandler` 实例来管理这个连接。`EchoServer` 和 `EchoHandler` 在不同的类中定义，因为它们完成不同的工作。`EchoServer` 接受一个连接时会建立一个新的套接字。并非在 `EchoServer` 中将这个套接字分派到单个客户，而是会创建一个 `EchoHandler`，从而充分利用由 `asyncore` 维护的套接字映射。

```
class EchoHandler(asyncore.dispatcher):
```

```

"""Handles echoing messages from a single client.
    """

def __init__(self, sock, chunk_size=256):
    self.chunk_size = chunk_size
    logger_name = 'EchoHandler'
    self.logger = logging.getLogger(logger_name)
    asyncore.dispatcher.__init__(self, sock=sock)
    self.data_to_write = []
    return

def writable(self):
    """Write if data has been received."""
    response = bool(self.data_to_write)
    self.logger.debug('writable() -> %s', response)
    return response

def handle_write(self):
    """Write as much as possible of the
    most recent message received.
    """
    data = self.data_to_write.pop()
    sent = self.send(data[:self.chunk_size])
    if sent < len(data):
        remaining = data[sent:]
        self.data_to_write.append(remaining)
    self.logger.debug('handle_write() -> (%d) %r',
                      sent, data[:sent])
    if not self.writable():
        self.handle_close()

def handle_read(self):
    """Read an incoming message from the client
    and put it into the outgoing queue.
    """
    data = self.recv(self.chunk_size)
    self.logger.debug('handle_read() -> (%d) %r',
                      len(data), data)
    self.data_to_write.insert(0, data)

def handle_close(self):
    self.logger.debug('handle_close()')
    self.close()

```

11.4.2 客户

要基于 `asyncore` 创建一个客户，需要派生 `dispatcher`，并提供实现来完成套接字创建和读写。对于 `EchoClient`，可以使用基类方法 `create_socket()` 在 `__init__()` 中创建套接字。也可以提

供这个方法的其他实现，不过在这里我们需要一个 TCP/IP 套接字，所以这个基类版本就足够了。

```
class EchoClient(asyncore.dispatcher):
    """Sends messages to the server and receives responses.
    """

    def __init__(self, host, port, message, chunk_size=128):
        self.message = message
        self.to_send = message
        self.received_data = []
        self.chunk_size = chunk_size
        self.logger = logging.getLogger('EchoClient')
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.logger.debug('connecting to %s', (host, port))
        self.connect((host, port))
        return
```

这里有 `handle_connect()` hook 只是为了显示它何时得到调用。其他类型的客户如果需要实现连接握手或协议协商，则应当在 `handle_connect()` 中完成这个协商工作。

```
def handle_connect(self):
    self.logger.debug('handle_connect()')
```

这里有 `handle_close()` 方法同样是为了显示它在处理期间何时得到调用。基类中的这个方法正确地关闭了套接字，所以如果应用不需要在关闭时做额外的清理工作，就不需要覆盖这个方法。

```
def handle_close(self):
    self.logger.debug('handle_close()')
    self.close()
    received_message = ''.join(self.received_data)
    if received_message == self.message:
        self.logger.debug('RECEIVED COPY OF MESSAGE')
    else:
        self.logger.debug('ERROR IN TRANSMISSION')
        self.logger.debug('EXPECTED "%s"', self.message)
        self.logger.debug('RECEIVED "%s"', received_message)
    return
```

`asyncore` 循环使用 `writable()` 和它的对应方法 `readable()` 来确定对每个分派器做何处理。对于各个分派器管理的套接字或文件描述符，`poll()` 或 `select()` 的具体使用就在 `asyncore` 代码中处理，不需要在程序中使用 `asyncore` 实现。这个程序只需要指示分派器希望读还是写数据。在这个客户中，只要有数据发送到服务器，`writable()` 就返回 `True`。`readable()` 总是返回 `True`，因为客户需要读取所有数据。

```
def writable(self):
    self.logger.debug('writable() -> %s', bool(self.to_send))
    return bool(self.to_send)
def readable(self):
```

```
self.logger.debug('readable() -> True')
return True
```

每次通过处理循环时，如果 `writable()` 做出肯定响应，就会调用 `handle_write()`。EchoClient 根据给定的大小限制将消息划分为多个部分，来展示一个相当大的多部分消息如何通过循环使用多次迭代进行传输。每次调用 `handle_write()` 时，会写下一部分消息，直至消息完全利用完毕。

```
def handle_write(self):
    sent = self.send(self.to_send[:self.chunk_size])
    self.logger.debug('handle_write() -> (%d) %r',
                      sent, self.to_send[:sent])
    self.to_send = self.to_send[sent:]
```

类似地，`readable()` 做出肯定响应时，说明有数据可以读取，则会调用 `handle_read()`。

```
def handle_read(self):
    data = self.recv(self.chunk_size)
    self.logger.debug('handle_read() -> (%d) %r',
                      len(data), data)
    self.received_data.append(data)
```

11.4.3 事件循环

这个模块中包括一个简短的测试脚本。它建立了一个服务器和客户，然后运行 `asyncore.loop()` 处理通信。创建客户时，会向由 `asyncore` 在内部维护的一个“映射”注册。随着循环迭代处理客户，开始出现通信。客户从一个看上去可读的套接字读取 0 字节时，这个条件会解释为连接关闭，相应地调用 `handle_close()`。

```
if __name__ == '__main__':
    import socket

    logging.basicConfig(level=logging.DEBUG,
                        format='%(name)-11s: %(message)s',
                        )

    address = ('localhost', 0) # let the kernel assign a port
    server = EchoServer(address)
    ip, port = server.address # find out which port was assigned
    message = open('lorem.txt', 'r').read()
    logging.info('Total message length: %d bytes', len(message))

    client = EchoClient(ip, port, message=message)

    asyncore.loop()
```

以下是运行这个程序的输出：

```
$ python asyncore_echo_server.py
```

```

EchoServer : binding to ('127.0.0.1', 63985)
root       : Total message length: 133 bytes
EchoClient : connecting to ('127.0.0.1', 63985)
EchoClient : readable() -> True
EchoClient : writable() -> True
EchoServer : handle_accept() -> ('127.0.0.1', 63986)
EchoServer : handle_close()
EchoClient : handle_connect()
EchoClient : handle_write() -> (128) 'Lorem ipsum dolor sit amet, cons
ectetuer adipiscing elit. Donec\negestas, enim et consectetur ullamco
rper, lectus ligula rutrum '
EchoClient : readable() -> True
EchoClient : writable() -> True
EchoHandler: writable() -> False
EchoHandler: handle_read() -> (128) 'Lorem ipsum dolor sit amet, conse
ctetuer adipiscing elit. Donec\negestas, enim et consectetur ullamcor
per, lectus ligula rutrum '
EchoClient : handle_write() -> (5) 'leo.\n'
EchoClient : readable() -> True
EchoClient : writable() -> False
EchoHandler: writable() -> True
EchoHandler: handle_read() -> (5) 'leo.\n'
EchoHandler: handle_write() -> (128) 'Lorem ipsum dolor sit amet, cons
ectetuer adipiscing elit. Donec\negestas, enim et consectetur ullamco
rper, lectus ligula rutrum '
EchoHandler: writable() -> True
EchoClient : readable() -> True
EchoClient : writable() -> False
EchoHandler: writable() -> True
EchoClient : handle_read() -> (128) 'Lorem ipsum dolor sit amet, conse
ctetuer adipiscing elit. Donec\negestas, enim et consectetur ullamcor
per, lectus ligula rutrum '
EchoHandler: handle_write() -> (5) 'leo.\n'
EchoHandler: writable() -> False
EchoHandler: handle_close()
EchoClient : readable() -> True
EchoClient : writable() -> False
EchoClient : handle_read() -> (5) 'leo.\n'
EchoClient : readable() -> True
EchoClient : writable() -> False
EchoClient : handle_close()
EchoClient : RECEIVED COPY OF MESSAGE
EchoClient : handle_read() -> (0) ''

```

在这个例子中，服务器、处理器和客户对象都在一个进程中由 `asyncore` 在同一个套接字映射中维护。为了区分服务器和客户，这里从不同的脚本分别实例化服务器和客户，并分别运行 `asyncore.loop()`。关闭一个分派器时，会把它从 `asyncore` 维护的映射中删除，映射为空时循环退出。

11.4.4 处理其他事件循环

有时有必要将 `asyncore` 事件循环与父应用的事件循环集成。例如，一个 GUI 应用可能不希望 UI 阻塞直到所有异步传输都处理完——这有违其异步的本来意图。为了轻松地实现这种集成，`asyncore.loop()` 接受一些参数，可以设置一个超时值，还可以限制循环运行的次数。可以用一个 HTTP 客户来说明这些选项对客户的影响，这是以 `asyncore` 标准库文档中的版本为基础构建的一个 HTTP 客户。

```
import asyncore
import logging
import socket
from cStringIO import StringIO
import urlparse

class HttpClient(asyncore.dispatcher):

    def __init__(self, url):
        self.url = url
        self.logger = logging.getLogger(self.url)
        self.parsed_url = urlparse.urlparse(url)
        asyncore.dispatcher.__init__(self)
        self.write_buffer = 'GET %s HTTP/1.0\r\n\r\n' % self.url
        self.read_buffer = StringIO()
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        address = (self.parsed_url.netloc, 80)
        self.logger.debug('connecting to %s', address)
        self.connect(address)

    def handle_connect(self):
        self.logger.debug('handle_connect()')

    def handle_close(self):
        self.logger.debug('handle_close()')
        self.close()

    def writable(self):
        is_writable = (len(self.write_buffer) > 0)
        if is_writable:
            self.logger.debug('writable() -> %s', is_writable)
        return is_writable

    def readable(self):
        self.logger.debug('readable() -> True')
        return True

    def handle_write(self):
        sent = self.send(self.write_buffer)
```

```

        self.logger.debug('handle_write() -> "%s"',
                           self.write_buffer[:sent])
        self.write_buffer = self.write_buffer[sent:]

    def handle_read(self):
        data = self.recv(8192)
        self.logger.debug('handle_read() -> %d bytes', len(data))
        self.read_buffer.write(data)

```

这个主程序在一个 while 循环中使用客户类，每次迭代时会读或写数据。

```

import asyncore
import logging

from asyncore_http_client import HttpClient
logging.basicConfig(level=logging.DEBUG,
                    format='%(name)s: %(message)s',
                    )

clients = [
    HttpClient('http://www.doughellmann.com/'),
]

loop_counter = 0
while asyncore.socket_map:
    loop_counter += 1
    logging.debug('loop_counter=%s', loop_counter)
    asyncore.loop(timeout=1, count=1)

```

可以不在一个定制局部 while 循环中调用 `asyncore.loop()`，而是以同样的方式从 GUI 工具包空闲处理器中调用，或者采用其他机制，在 UI 不忙于其他事件处理程序时完成少量工作。

```
$ python asyncore_loop.py
```

```

http://www.doughellmann.com/: connecting to ('www.doughellmann.com',
80)
root: loop_counter=1
http://www.doughellmann.com/: readable() -> True
http://www.doughellmann.com/: writable() -> True
http://www.doughellmann.com/: handle_connect()
http://www.doughellmann.com/: handle_write() -> "GET http://www.doug
hellmann.com/ HTTP/1.0

"
root: loop_counter=2
http://www.doughellmann.com/: readable() -> True
http://www.doughellmann.com/: handle_read() -> 1448 bytes
root: loop_counter=3
http://www.doughellmann.com/: readable() -> True

```

```

http://www.doughellmann.com/: handle_read() -> 2896 bytes
root: loop_counter=4
http://www.doughellmann.com/: readable() -> True
http://www.doughellmann.com/: handle_read() -> 1318 bytes
root: loop_counter=5
http://www.doughellmann.com/: readable() -> True
http://www.doughellmann.com/: handle_close()
http://www.doughellmann.com/: handle_read() -> 0 bytes

```

11.4.5 处理文件

正常情况下，`asyncore` 用于套接字，不过有时对于异步地读文件也很有用（例如，测试网络服务器时，要使用文件而不需要网络设置，或者需要分部分读写大的数据文件）。对于这些情况，`asyncore` 提供了 `file_dispatcher` 和 `file_wrapper` 类。

下面这个例子实现了文件的一个异步阅读器，每次调用 `handle_read()` 时会响应另一部分数据。

```

class FileReader(asyncore.file_dispatcher):

    def writable(self):
        return False

    def handle_read(self):
        data = self.recv(64)
        print 'READ: (%d)\n%x' % (len(data), data)

    def handle_expt(self):
        # Ignore events that look like out of band data
        pass

    def handle_close(self):
        self.close()

```

要使用 `FileReader()`，需要提供一个打开的文件句柄，作为构造函数的惟一参数。

```

reader = FileReader(open('lorem.txt', 'r'))
asyncore.loop()

```

注意：这个例子在 Python 2.7 下通过测试。对于 Python 2.5 和更早版本，`file_dispatcher` 不会自动将一个打开的文件转换为文件描述符，而是要使用 `os.popen()` 打开文件，并把描述符传入 `FileReader`。

运行这个程序会生成以下输出：

```

$ python asyncore_file_dispatcher.py
READ: (64)
'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec\n'

```

```

READ: (64)
'egestas, enim et consectetur ullamcorper, lectus ligula rutrum '
READ: (5)
'leo.\n'
READ: (0)
''

```

参见:

`asyncore` (<http://docs.python.org/library/asyncore.html>) 这个模块的标准库文档。

`asynchat` (11.5 节) `asynchat` 模块建立在 `asyncore` 基础上, 提供了一个实现协议的框架, 其出发点是使用一个集合协议来回传递消息。

`SocketServer` (11.3 节) `SocketServer` 模块一节包括另一个使用多线程和进程实现的 `EchoServer` 例子。

11.5 asynchat——异步协议处理器

作用: 异步网络通信协议处理器。

Python 版本: 1.5.2 及以后版本

`asynchat` 模块建立在 `asyncore` 基础上, 为实现通信协议提供了一个框架, 其出发点是在服务器和客户之间来回传递消息。`async_chat` 类是一个 `asyncore.dispatcher` 子类, 它接收数据, 并查找一个消息终止符。这个子类只需要指定数据到来时要做什么, 以及一旦发现终止符该如何响应。发出的数据会排队, 等待通过由 `async_chat` 管理的 FIFO 对象传输。

11.5.1 消息终止符

到来的消息根据终止符 (terminator) 分解, 要通过 `set_terminator()` 为每一个 `async_chat` 实例管理终止符。有 3 种可能的配置。

1. 如果向 `set_terminator()` 传递一个字符串参数, 当输入数据中出现这个字符串时, 则认为这个消息是完整的。
2. 如果传入一个数值参数, 当读入指定字节数的数据时则认为消息是完整的。
3. 如果传入 `None`, 消息的终止不由 `async_chat` 管理。

下一个 `EchoServer` 例子使用了一个简单的字符串终止符和一个消息长度终止符, 这取决于到来数据的上下文。标准库文档中的 HTTP 请求处理器示例还提供了另一个例子, 来说明如何根据上下文改变终止符。它在读取 HTTP 首部时使用了一个字面量终止符, 另外使用一个长度值来终止 HTTP POST 请求体。

11.5.2 服务器和处理器

为了更易于理解 `asynchat` 与 `asyncore` 有什么区别, 这里的例子重复了讨论 `asyncore` 时 `EchoServer` 例子的功能。这个例子仍需要同样的部分: 一个服务器对象来接受连接, 一些处理器对象处理与各个客户的通信, 以及一些客户对象来初始化通信。

使用 `asynchat` 的 `EchoServer` 实现实际上与 `asyncore` 例子中创建的服务器是相同的，只不过日志记录调用更少：

```
import asyncore
import logging
import socket

from asynchat_echo_handler import EchoHandler

class EchoServer(asyncore.dispatcher):
    """Receives connections and establishes handlers for each client.
    """

    def __init__(self, address):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(address)
        self.address = self.socket.getsockname()
        self.listen(1)
        return

    def handle_accept(self):
        # Called when a client connects to our socket
        client_info = self.accept()
        EchoHandler(sock=client_info[0])
        # Only deal with one client at a time,
        # so close as soon as the handler is set up.
        # Under normal conditions, the server
        # would run forever or until it received
        # instructions to stop.
        self.handle_close()
        return

    def handle_close(self):
        self.close()
```

这个版本的 `EchoHandler` 建立在 `asynchat.async_chat` 基础上，而不是 `asyncore.dispatcher`。它在一个稍高的抽象层次上操作，读写会自动处理。缓冲区需要了解4个方面：

- 如何处理到来的数据（覆盖 `handle_incoming_data()`）
- 如何识别到来消息的结束（通过 `set_terminator()`）
- 接收到一个完整消息时做什么（`found_terminator()` 中）
- 要发送什么数据（使用 `push()`）

这个示例应用有两种操作模式。它会等待 `ECHO length\n` 形式的命令，或者等待数据回应。通过将一个实例变量 `process_data` 设置为发现终止符时所要调用的方法，然后在适当时改变终止符，可以来回切换操作模式。

```
import asynchat
import logging

class EchoHandler(asynchat.async_chat):
    """Handles echoing messages from a single client.
    """

    # Artificially reduce buffer sizes to illustrate
    # sending and receiving partial messages.
    ac_in_buffer_size = 128
    ac_out_buffer_size = 128

    def __init__(self, sock):
        self.received_data = []
        self.logger = logging.getLogger('EchoHandler')
        asynchat.async_chat.__init__(self, sock)
        # Start looking for the ECHO command
        self.process_data = self._process_command
        self.set_terminator('\n')
        return

    def collect_incoming_data(self, data):
        """Read an incoming message from the client
        and put it into the outgoing queue.
        """
        self.logger.debug(
            'collect_incoming_data() -> (%d bytes) %r',
            len(data), data)
        self.received_data.append(data)

    def found_terminator(self):
        """The end of a command or message has been seen."""
        self.logger.debug('found_terminator()')
        self.process_data()

    def _process_command(self):
        """Have the full ECHO command"""
        command = ''.join(self.received_data)
        self.logger.debug('_process_command() %r', command)
        command_verb, command_arg = command.strip().split(' ')
        expected_data_len = int(command_arg)
        self.set_terminator(expected_data_len)
        self.process_data = self._process_message
        self.received_data = []

    def _process_message(self):
        """Have read the entire message."""
        to_echo = ''.join(self.received_data)
        self.logger.debug('_process_message() echoing %r',
```

```

        to_echo)
    self.push(to_echo)
    # Disconnect after sending the entire response
    # since we only want to do one thing at a time
    self.close_when_done()

```

一旦发现完整命令，处理程序会切换为消息处理模式，等待接收完整的文本。所有数据都得到时，会推至发出通道。数据一旦发送就会关闭处理程序。

11.5.3 客户

客户与处理程序的做法相同。类似于 `asyncore` 实现，要发送的消息是客户构造函数的一个参数。建立套接字连接时，会调用 `handle_connect()`，使客户能够发送命令和消息数据。

命令直接推送，但是对于消息文本，会使用一个特殊的“生成器”类。将轮询这个生成器，通过网络发出数据块。生成器返回一个空串时，则认为它为空，写操作结束。

客户希望只响应消息数据，所以它设置了一个整数终止符，在一个列表中收集数据，直至接收到整个消息。

```

import asynchat
import logging
import socket

class EchoClient(asynchat.async_chat):
    """Sends messages to the server and receives responses.
    """

    # Artificially reduce buffer sizes to show
    # sending and receiving partial messages.
    ac_in_buffer_size = 128
    ac_out_buffer_size = 128

    def __init__(self, host, port, message):
        self.message = message
        self.received_data = []
        self.logger = logging.getLogger('EchoClient')
        asynchat.async_chat.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.logger.debug('connecting to %s', (host, port))
        self.connect((host, port))
        return

    def handle_connect(self):
        self.logger.debug('handle_connect()')
        # Send the command
        self.push('ECHO %d\n' % len(self.message))
        # Send the data

```

```

        self.push_with_producer(
            EchoProducer(self.message,
                          buffer_size=self.ac_out_buffer_size)
        )
        # We expect the data to come back as-is,
        # so set a length-based terminator
        self.set_terminator(len(self.message))
    def collect_incoming_data(self, data):
        """Read an incoming message from the client
        and add it to the outgoing queue.
        """
        self.logger.debug(
            'collect_incoming_data() -> (%d) %r',
            len(data), data)
        self.received_data.append(data)

    def found_terminator(self):
        self.logger.debug('found_terminator()')
        received_message = ''.join(self.received_data)
        if received_message == self.message:
            self.logger.debug('RECEIVED COPY OF MESSAGE')
        else:
            self.logger.debug('ERROR IN TRANSMISSION')
            self.logger.debug('EXPECTED %r', self.message)
            self.logger.debug('RECEIVED %r', received_message)
        return

class EchoProducer(asyncchat.simple_producer):

    logger = logging.getLogger('EchoProducer')

    def more(self):
        response = asyncchat.simple_producer.more(self)
        self.logger.debug('more() -> (%s bytes) %r',
                          len(response), response)
        return response

```

11.5.4 集成

这个例子的主程序在同一个 `asyncore` 主循环中建立了客户端和服务端。

```

import asyncore
import logging
import socket

from asyncchat_echo_server import EchoServer
from asyncchat_echo_client import EchoClient

```



```

logging.basicConfig(level=logging.DEBUG,
                    format='%(name)-11s: %(message)s',
                    )

address = ('localhost', 0) # let the kernel give us a port
server = EchoServer(address)
ip, port = server.address # find out what port we were given

message_data = open('lorem.txt', 'r').read()
client = EchoClient(ip, port, message=message_data)

asyncore.loop()

```

正常情况下会在单独的进程中运行服务器和客户, 不过采用这里的做法更容易显示组合的输出。

```
$ python asynchat_echo_main.py
```

```

EchoClient : connecting to ('127.0.0.1', 52590)
EchoClient : handle_connect()
EchoProducer: more() -> (128 bytes) 'Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Donec\negestas, enim et consectetue
r ullamcorper, lectus ligula rutrum\n'
EchoProducer: more() -> (38 bytes) 'leo, a elementum elit tortor
eu quam.\n'
EchoProducer: more() -> (0 bytes) ''
EchoHandler: collect_incoming_data() -> (8 bytes) 'ECHO 166'
EchoHandler: found_terminator()
EchoHandler: _process_command() 'ECHO 166'
EchoHandler: collect_incoming_data() -> (119 bytes) 'Lorem ipsum
dolor sit amet, consectetuer adipiscing elit. Donec\negestas, eni
m et consectetuer ullamcorper, lectus ligul'
EchoHandler: collect_incoming_data() -> (47 bytes) 'a rutrum\nleo
, a elementum elit tortor eu quam.\n'
EchoHandler: found_terminator()
EchoHandler: _process_message() echoing 'Lorem ipsum dolor sit am
et, consectetuer adipiscing elit. Donec\negestas, enim et consect
etuer ullamcorper, lectus ligula rutrum\nleo, a elementum elit to
rtor eu quam.\n'
EchoClient : collect_incoming_data() -> (128) 'Lorem ipsum dolor
sit amet, consectetuer adipiscing elit. Donec\negestas, enim et c
onsectetuer ullamcorper, lectus ligula rutrum\n'
EchoClient : collect_incoming_data() -> (38) 'leo, a elementum el
it tortor eu quam.\n'
EchoClient : found_terminator()
EchoClient : RECEIVED COPY OF MESSAGE

```

参见:

[asynchat \(http://docs.python.org/library/asynchat.html\)](http://docs.python.org/library/asynchat.html) 这个模块的标准库文档。

[asyncore \(11.4 节\)](#) `asyncore` 模块实现了一个底层异步 I/O 事件循环。

第 12 章

Internet

Internet 是现代计算很普及的一个领域。甚至很小的、单一用途的脚本也会频繁与远程服务交互，发送或接收数据。Python 提供了一组丰富的工具来处理 Web 协议，因此非常适合编写基于 Web 的应用（不论作为客户还是服务器）。

`urlparse` 模块管理 URL 串，可以分解或组合 URL 串的组成部分，在客户和服务端中很有用。

有两个客户端 API 可以用来处理 Web 资源。原来的 `urllib` 和更新的 `urllib2` 提供了类似的 API 来远程获取内容，不过 `urllib2` 更容易扩展新协议，而且 `urllib2.Request` 提供了一种方法，可以向发出的请求增加定制首部。

HTTP POST 请求通常用 `urllib` 完成“表单编码”。通过 POST 发送的二进制数据应当首先用 Base64 编码，以符合消息格式标准。

合法客户作为 spider（蜘蛛）或 crawler（爬虫）^① 访问大量网站时，应当使用 `robotparser`，从而在对远程服务器带来过重负载之前能够确保确实有权限。

要用 Python 创建一个定制 Web 服务器，而不需要任何外部框架，可以使用 `BaseHTTPServer` 作为起点。它会处理 HTTP 协议，所以惟一需要的定制就是响应到来请求的应用代码。

服务器中的会话状态可以通过 `cookie` 来管理，`cookie` 由 `Cookie` 模块创建和解析。由于 `Cookie` 模块提供了对 `cookie` 到期、路径、域以及其他设置的完全支持，因此很容易配置会话。

`uuid` 模块用于为需要惟一值的资源生成标识符。UUID 很适合自动生成统一资源名（Uniform Resource Name, URN）值，其中资源名必须惟一，但并不需要表达任何具体含义。

Python 的标准库支持两个基于 Web 的远程过程调用机制。AJAX 通信中使用的 JavaScript 对象记法（JavaScript Object Notation, JSON）编码机制在 `json` 中实现。它在客户或服务端中同样适用。另外 `xmlrpclib` 和 `SimpleXMLRPCServer` 中分别包含了完整的 XML-RPC 客户和服务端库。

12.1 urlparse——分解 URL

作用：将 URL 分解为其组成部分。

Python 版本：1.4 及以后版本

① 蜘蛛（spider）或爬虫（crawler）是指一段计算机程序，它从互联网上按照一定的逻辑和算法抓取并下载互联网的网页，是搜索引擎的一个重要组成部分。——译者注

urlparse 模块提供了一些函数，可以按相关 RFC 定义将 URL 分解为其组成部分。

12.1.1 解析

urlparse() 函数的返回值是一个对象，相当于一个包含 6 个元素的 tuple。

```
from urlparse import urlparse

url = 'http://netloc/path;param?query=arg#frag'
parsed = urlparse(url)
print parsed
```

通过元组接口得到的 URL 部分分别是机制、网络位置、路径、路径段参数（由一个分号与路径分开）、查询以及片段。

```
$ python urlparse_urlparse.py

ParseResult(scheme='http', netloc='netloc', path='/path',
params='param', query='query=arg', fragment='frag')
```

尽管返回值相当于一个元组，但实际上它基于一个 namedtuple，这是 tuple 的一个子类，除了可以通过索引访问，它还支持通过命名属性访问 URL 的各部分。属性 API 不仅更易于程序员使用，还允许访问 tuple API 未提供的很多值。

```
from urlparse import urlparse

url = 'http://user:pwd@NetLoc:80/path;param?query=arg#frag'
parsed = urlparse(url)
print 'scheme :', parsed.scheme
print 'netloc :', parsed.netloc
print 'path :', parsed.path
print 'params :', parsed.params
print 'query :', parsed.query
print 'fragment:', parsed.fragment
print 'username:', parsed.username
print 'password:', parsed.password
print 'hostname:', parsed.hostname, '(netloc in lowercase)'
print 'port :', parsed.port
```

输入 URL 中有用户名 (username) 和密码 (password) 时，由 urlparse() 可以得到，如果没有提供用户名和密码则设置为 None。主机名 (hostname) 与 netloc 值相同，全为小写。如果有端口 (port) 则转换为一个整数，如果没有会设置为 None。

```
$ python urlparse_urlparseattrs.py

scheme : http
netloc : user:pwd@NetLoc:80
path : /path
```

```

params : param
query : query=arg
fragment: frag
username: user
password: pwd
hostname: netloc (netloc in lowercase)
port : 80

```

urlsplit() 函数可以替换 urlparse(), 但表现稍有不同, 因为它不会从 URL 分解参数。这对于遵循 RFC 2396 的 URL 很有用, 它支持对应路径每一段的参数。

```

from urlparse import urlsplit

url = 'http://user:pwd@NetLoc:80/p1;param/p2;param?query=arg#frag'
parsed = urlsplit(url)
print parsed
print 'scheme :', parsed.scheme
print 'netloc :', parsed.netloc
print 'path :', parsed.path
print 'query :', parsed.query
print 'fragment:', parsed.fragment
print 'username:', parsed.username
print 'password:', parsed.password
print 'hostname:', parsed.hostname, '(netloc in lowercase)'
print 'port :', parsed.port

```

由于没有分解参数, tuple API 会显示 5 个元素而不是 6 个, 这里没有 params 属性。

```
$ python urlparse_urlsplit.py
```

```

SplitResult(scheme='http', netloc='user:pwd@NetLoc:80',
path='/p1;param/p2;param', query='query=arg', fragment='frag')
scheme : http
netloc : user:pwd@NetLoc:80
path : /p1;param/p2;param
query : query=arg
fragment: frag
username: user
password: pwd
hostname: netloc (netloc in lowercase)
port : 80

```

要想从一个 URL 剥离出片段标识符, 如从一个 URL 查找基页面名, 可以使用 urldefrag()。

```

from urlparse import urldefrag

original = 'http://netloc/path;param?query=arg#frag'
print 'original:', original
url, fragment = urldefrag(original)

```

```
print 'url      :', url
print 'fragment:', fragment
```

返回值是一个元组，其中包含基 URL 和片段。

```
$ python urlparse_urldefrag.py
```

```
original: http://netloc/path;param?query=arg#frag
url      : http://netloc/path;param?query=arg
fragment: frag
```

12.1.2 反解析

还可以利用一些方法把分解的 URL 的各个部分重新组装在一起，形成一个串。解析的 URL 对象有一个 `geturl()` 方法。

```
from urlparse import urlparse
```

```
original = 'http://netloc/path;param?query=arg#frag'
print 'ORIG  :', original
parsed = urlparse(original)
print 'PARSED:', parsed.geturl()
```

`geturl()` 只适用于 `urlparse()` 或 `urlsplit()` 返回的对象。

```
$ python urlparse_geturl.py
```

```
ORIG  : http://netloc/path;param?query=arg#frag
PARSED: http://netloc/path;param?query=arg#frag
```

利用 `urlunparse()` 可以将包含串的普通元组重新组合为一个 URL。

```
from urlparse import urlparse, urlunparse
```

```
original = 'http://netloc/path;param?query=arg#frag'
print 'ORIG  :', original
parsed = urlparse(original)
print 'PARSED:', type(parsed), parsed
t = parsed[:]
print 'TUPLE  :', type(t), t
print 'NEW    :', urlunparse(t)
```

尽管 `urlparse()` 返回的 `ParseResult` 可以用作一个元组，不过这个例子显式地创建了一个新元组，来展示 `urlunparse()` 也适用于普通元组。

```
$ python urlparse_urlunparse.py
```

```
ORIG  : http://netloc/path;param?query=arg#frag
PARSED: <class 'urlparse.ParseResult'> ParseResult(scheme='http',
netloc='netloc', path='/path', params='param', query='query=arg',
fragment='frag')
```

```
TUPLE : <type 'tuple'> ('http', 'netloc', '/path', 'param',
'query=arg', 'frag')
NEW   : http://netloc/path;param?query=arg#frag
```

如果输入 URL 包含多余的部分, 重新构造的 URL 可能会将其去除。

```
from urlparse import urlparse, urlunparse
```

```
original = 'http://netloc/path;?#'
print 'ORIG  :', original
parsed = urlparse(original)
print 'PARSED:', type(parsed), parsed
t = parsed[:]
print 'TUPLE :', type(t), t
print 'NEW   :', urlunparse(t)
```

在这里, 原 URL 中缺少参数、查询和片段。新 URL 看起来与原 URL 并不相同, 不过按照标准它们是等价的。

```
$ python urlparse_urlunparseextra.py
```

```
ORIG  : http://netloc/path;?#
PARSED: <class 'urlparse.ParseResult'> ParseResult(scheme='http',
netloc='netloc', path='/path', params='', query='', fragment='')
TUPLE : <type 'tuple'> ('http', 'netloc', '/path', '', '', '')
NEW   : http://netloc/path
```

12.1.3 连接

除了解析 URL 外, `urlparse` 还包括一个 `urljoin()` 方法, 可以由相对片段构造绝对 URL。

```
from urlparse import urljoin
```

```
print urljoin('http://www.example.com/path/file.html',
              'anotherfile.html')
print urljoin('http://www.example.com/path/file.html',
              '../anotherfile.html')
```

在这个例子中, 计算第 2 个 URL 时要考虑路径的相对部分 (“../”)。

```
$ python urlparse_urljoin.py
```

```
http://www.example.com/path/anotherfile.html
http://www.example.com/anotherfile.html
```

非相对路径的处理与 `os.path.join()` 的处理方式相同。

```
from urlparse import urljoin
```

```
print urljoin('http://www.example.com/path/',
              '/subpath/file.html')
```

```
print urljoin('http://www.example.com/path/',
              'subpath/file.html')
```

如果连接到 URL 的路径以一个斜线开头 (/), 这会将 URL 的路径重置为顶级路径。如果不是以一个斜线开头, 则追加到当前 URL 路径的末尾。

```
$ python urlparse_urljoin_with_path.py

http://www.example.com/subpath/file.html
http://www.example.com/path/subpath/file.html
```

参见:

`urlparse` (<http://docs.python.org/lib/module-urlparse.html>) 这个模块的标准库文档。

`urllib` (12.3 节) 获取一个 URL 标识的资源的内容。

`urllib2` (12.4 节) 访问远程 URL 的候选 API。

RFC 1738 (<http://tools.ietf.org/html/rfc1738.html>) 统一资源定位符 (URL) 语法。

RFC 1808 (<http://tools.ietf.org/html/rfc1808.html>) 相对 URL。

RFC 2396 (<http://tools.ietf.org/html/rfc2396.html>) 统一资源标识符 (URI) 通用语法。

RFC 3986 (<http://tools.ietf.org/html/rfc3986.html>) 统一资源标识符 (URI) 语法。

12.2 BaseHTTPServer——实现 Web 服务器的基类

作用: `BaseHTTPServer` 包含一些类, 可以构成 Web 服务器的基础。

Python 版本: 1.4 及以后版本

`BaseHTTPServer` 使用 `SocketServer` 的类创建基类, 用来建立 HTTP 服务器。`HTTPServer` 可以直接使用, 不过 `BaseHTTPRequestHandler` 需要扩展来处理各个协议方法 (GET、POST 等等)。

12.2.1 HTTP GET

要在一个请求处理器类中添加一个 HTTP 方法支持, 需要实现方法 `do_METHOD()`, 这里的 `METHOD` 要替换为具体的 HTTP 方法名 (例如, `do_GET()`、`do_POST()` 等等)。为保持一致, 请求处理器方法不带任何参数。请求的所有参数都由 `BaseHTTPRequestHandler` 解析, 并存储为请求实例的实例属性。

下面这个示例请求处理器展示了如何向客户返回一个响应, 其中包含对构建响应可能有用的一些本地属性。

```
from BaseHTTPServer import BaseHTTPRequestHandler
import urlparse

class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        parsed_path = urlparse.urlparse(self.path)
```

```

message_parts = [
    'CLIENT VALUES:',
    'client_address=%s (%s)' % (self.client_address,
                                self.address_string()),
    'command=%s' % self.command,
    'path=%s' % self.path,
    'real path=%s' % parsed_path.path,
    'query=%s' % parsed_path.query,
    'request_version=%s' % self.request_version,
    '',
    'SERVER VALUES:',
    'server_version=%s' % self.server_version,
    'sys_version=%s' % self.sys_version,
    'protocol_version=%s' % self.protocol_version,
    '',
    'HEADERS RECEIVED:',
]
for name, value in sorted(self.headers.items()):
    message_parts.append('%s=%s' % (name, value.rstrip()))
message_parts.append('')
message = '\r\n'.join(message_parts)
self.send_response(200)
self.end_headers()
self.wfile.write(message)
return

if __name__ == '__main__':
    from BaseHTTPServer import HTTPServer
    server = HTTPServer(('localhost', 8080), GetHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()

```

先组装消息文本，然后写至 `wfile`，这是包装了响应套接字的文件句柄。每个响应需要一个响应码，通过 `send_response()` 设置。如果使用了一个错误码（404, 501 等等），首部会包含一个相应的默认错误消息，或者可能会随这个错误码传递一个消息。

要在服务器中运行请求处理器，需要将它传递到 `HTTPServer` 的构造函数，如示例脚本中 `__main__` 处理部分所示。

然后启动服务器。

```
$ python BaseHTTPServer_GET.py
```

```
Starting server, use <Ctrl-C> to stop
```

在另外一个终端使用 `curl` 来访问这个服务器。

```
$ curl -i http://localhost:8080/?foo=bar
```

```
HTTP/1.0 200 OK
```



```

Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 09 Dec 2007 16:00:34 GMT

CLIENT VALUES:
client_address=('127.0.0.1', 51275) (localhost)
command=GET
path=/?foo=bar
real_path=/
query=foo=bar
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0

```

12.2.2 HTTP POST

支持 POST 请求需要多做一些工作，因为基类不会自动解析表单数据。cgi 模块提供了 FieldStorage 类，如果给定了正确的输入，它知道如何解析表单。

```

from BaseHTTPServer import BaseHTTPRequestHandler
import cgi

class PostHandler(BaseHTTPRequestHandler):

    def do_POST(self):
        # Parse the form data posted
        form = cgi.FieldStorage(
            fp=self.rfile,
            headers=self.headers,
            environ={
                'REQUEST_METHOD': 'POST',
                'CONTENT_TYPE': self.headers['Content-Type'],
            })

        # Begin the response
        self.send_response(200)
        self.end_headers()
        self.wfile.write('Client: %s\n' % str(self.client_address))
        self.wfile.write('User-agent: %s\n' %
            str(self.headers['user-agent']))
        self.wfile.write('Path: %s\n' % self.path)
        self.wfile.write('Form data:\n')

        # Echo back information about what was posted in the form
        for field in form.keys():
            field_item = form[field]

```

```

        if field_item.filename:
            # The field contains an uploaded file
            file_data = field_item.file.read()
            file_len = len(file_data)
            del file_data
            self.wfile.write(
                '\tUploaded %s as "%s" (%d bytes)\n' % \
                (field, field_item.filename, file_len))
        else:
            # Regular form value
            self.wfile.write('\t%s=%s\n' %
                              (field, form[field].value))

    return

if __name__ == '__main__':
    from BaseHTTPServer import HTTPServer
    server = HTTPServer(('localhost', 8080), PostHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()

```

在一个窗口中运行这个服务器。

```
$ python BaseHTTPServer_POST.py
```

```
Starting server, use <Ctrl-C> to stop
```

通过使用 -F 选项, curl 的参数可以包括将提交给服务器的表单数据。最后一个参数 (-F datafile=@BaseHTTPServer_GET.py) 将提交文件 BaseHTTPServer_GET.py 的内容, 来展示如何从表单读取文件数据。

```

$ curl http://localhost:8080/ -F name=dhellmann -F foo=bar \
-F datafile=@BaseHTTPServer_GET.py

Client: ('127.0.0.1', 65029)
User-agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7
OpenSSL/0.9.8l zlib/1.2.3
Path: /
Form data:
  Uploaded datafile as "BaseHTTPServer_GET.py" (2580 bytes)
  foo=bar
  name=dhellmann

```

12.2.3 线程与进程

HTTPServer 是 SocketServer.TCPServer 的一个简单子类, 并不使用多线程或多进程来处理请求。要增加线程或进程, 需要使用相应的 mix-in 技术从 SocketServer 创建一个新类。

```

from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
from SocketServer import ThreadingMixIn
import threading

class Handler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        message = threading.currentThread().getName()
        self.wfile.write(message)
        self.wfile.write('\n')
        return

class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    """Handle requests in a separate thread."""

if __name__ == '__main__':
    server = ThreadedHTTPServer(('localhost', 8080), Handler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()

```

类似于其他例子，采用同样的方式运行服务器。

```
$ python BaseHTTPServer_threads.py
```

```
Starting server, use <Ctrl-C> to stop
```

每次服务器接收到一个请求时，它会开始一个新线程或进程来处理这个请求。

```
$ curl http://localhost:8080/
```

```
Thread-1
```

```
$ curl http://localhost:8080/
```

```
Thread-2
```

```
$ curl http://localhost:8080/
```

```
Thread-3
```

用 `ForkingMixIn` 替换 `ThreadingMixIn` 会得到类似的结果，不过会使用单独的进程而不是线程。

12.2.4 处理错误

处理错误时要调用 `send_error()`，并传入适当的错误码和一个可选的错误消息。整个响应（包括首部、状态码和响应体）会自动生成。

```

from BaseHTTPServer import BaseHTTPRequestHandler

class ErrorHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_error(404)
        return

if __name__ == '__main__':
    from BaseHTTPServer import HTTPServer
    server = HTTPServer(('localhost', 8080), ErrorHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()

```

在这里，会返回一个 404 错误。

```
$ python BaseHTTPServer_errors.py
```

```
Starting server, use <Ctrl-C> to stop
```

这里使用一个 HTML 文档将错误消息报告给客户，并提供一个首部指示错误码。

```
$ curl -i http://localhost:8080/
```

```

HTTP/1.0 404 Not Found
Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 09 Dec 2007 15:49:44 GMT
Content-Type: text/html
Connection: close

```

```

<head>
<title>Error response</title>
</head>
<body>
<h1>Error response</h1>
<p>Error code 404.
<p>Message: Not Found.
<p>Error code explanation: 404 = Nothing matches the given URI.
</body>

```

12.2.5 设置首部

`send_header` 方法将向 HTTP 响应添加首部数据。这个方法有两个参数：首部名和值。

```

from BaseHTTPServer import BaseHTTPRequestHandler
import urlparse
import time

class GetHandler(BaseHTTPRequestHandler):

```

```

def do_GET(self):
    self.send_response(200)
    self.send_header('Last-Modified',
                     self.date_time_string(time.time()))
    self.end_headers()
    self.wfile.write('Response body\n')
    return

if __name__ == '__main__':
    from BaseHTTPServer import HTTPServer
    server = HTTPServer(('localhost', 8080), GetHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()

```

这个例子将 Last-Modified 首部设置为当前时间戳（按照 RFC 2822 格式化）。

```
$ curl -i http://localhost:8080/
```

```

HTTP/1.0 200 OK
Server: BaseHTTP/0.3 Python/2.7
Date: Sun, 10 Oct 2010 13:58:32 GMT
Last-Modified: Sun, 10 Oct 2010 13:58:32 -0000

```

Response body

类似于其他例子，服务器将请求记录到终端。

```
$ python BaseHTTPServer_send_header.py
```

```
Starting server, use <Ctrl-C> to stop
```

参见：

BaseHTTPServer (<http://docs.python.org/library/basehttpserver.html>) 这个模块的标准库文档。

SocketServer (11.3 节) SocketServer 模块提供了处理原始套接字连接的基类。

RFC 2822 (<http://tools.ietf.org/html/rfc2822.html>) “Internet 消息格式”为基于文本的消息指定了一个格式，如 email 和 HTTP 响应。

12.3 urllib——网络资源访问

作用：访问不需要验证的远程资源、cookie 等等。

Python 版本：1.4 及以后版本

urllib 模块为网络资源访问提供了一个简单的接口。它还包括一些函数用于对参数编码和加引号，以便通过 HTTP 传递到一个服务器。

12.3.1 利用缓存实现简单获取

下载数据是一个很常见的操作，urllib 提供了 `urlretrieve()` 函数来满足这个需要。`urlretrieve()` 的参数包括一个 URL、存放数据的一个临时文件和一个报告下载进度的函数，另外如果 URL 指示一个表单，要求提交数据，那么 `urlretrieve()` 还要有一个参数表示要传递的数据。如果没有给定文件名，`urlretrieve()` 会创建一个临时文件。调用程序可以直接删除这个文件，或者将这个文件作为一个缓存，使用 `urlcleanup()` 将其删除。

下面这个例子使用一个 HTTP GET 请求从一个 Web 服务器获取数据。

```
import urllib
import os

def reporthook(blocks_read, block_size, total_size):
    """total_size is reported in bytes.
    block_size is the amount read each time.
    blocks_read is the number of blocks successfully read.
    """
    if not blocks_read:
        print 'Connection opened'
        return
    if total_size < 0:
        # Unknown size
        print 'Read %d blocks (%d bytes)' % (blocks_read,
                                             blocks_read * block_size)
    else:
        amount_read = blocks_read * block_size
        print 'Read %d blocks, or %d/%d' % \
              (blocks_read, amount_read, total_size)
    return

try:
    filename, msg = urllib.urlretrieve(
        'http://blog.doughellmann.com/',
        reporthook=reporthook)
    print
    print 'File:', filename
    print 'Headers:'
    print msg
    print 'File exists before cleanup:', os.path.exists(filename)

finally:
    urllib.urlcleanup()

    print 'File still exists:', os.path.exists(filename)
```

每次从服务器读取数据时，会调用 `reporthook()` 报告下载进度。这个函数的 3 个参数分

别是目前为止读取的块数、块的大小（字节数），以及正在下载的资源的大小（字节数）。服务器没有返回 Content-length 首部时，`urlretrieve()` 不知道数据有多大，为 `total_size` 参数传入 -1。

```
$ python urllib_urlretrieve.py
```

```
Connection opened
```

```
Read 1 blocks (8192 bytes)
```

```
Read 2 blocks (16384 bytes)
```

```
Read 3 blocks (24576 bytes)
```

```
Read 4 blocks (32768 bytes)
```

```
Read 5 blocks (40960 bytes)
```

```
Read 6 blocks (49152 bytes)
```

```
Read 7 blocks (57344 bytes)
```

```
Read 8 blocks (65536 bytes)
```

```
Read 9 blocks (73728 bytes)
```

```
Read 10 blocks (81920 bytes)
```

```
Read 11 blocks (90112 bytes)
```

```
Read 12 blocks (98304 bytes)
```

```
File: /var/folders/9R/9Rlt+tR02Raxzk+F71Q50U+++Uw/-Tmp-/tmpYI9AuC
```

```
Headers:
```

```
Content-Type: text/html; charset=UTF-8
```

```
Expires: Fri, 07 Jan 2011 14:23:06 GMT
```

```
Date: Fri, 07 Jan 2011 14:23:06 GMT
```

```
Last-Modified: Tue, 04 Jan 2011 12:32:04 GMT
```

```
ETag: "f2108552-7c52-4c50-8838-8300645c40be"
```

```
X-Content-Type-Options: nosniff
```

```
X-XSS-Protection: 1; mode=block
```

```
Server: GSE
```

```
Cache-Control: public, max-age=0, proxy-revalidate, must-revalidate
```

```
Age: 0
```

```
File exists before cleanup: True
```

```
File still exists: False
```

12.3.2 参数编码

可以对参数编码并追加到 URL，从而将它们传递到服务器。

```
import urllib
```

```
query_args = { 'q': 'query string', 'foo': 'bar' }
```

```
encoded_args = urllib.urlencode(query_args)
```

```
print 'Encoded:', encoded_args
```

```
url = 'http://localhost:8080/?' + encoded_args
```



```
print urllib.urlopen(url).read()
```

客户值列表中，查询包含了已编码的查询参数。

```
$ python urllib_urllencode.py
```

```
Encoded: q=query+string&foo=bar
CLIENT VALUES:
client_address=('127.0.0.1', 54415) (localhost)
command=GET
path=?q=query+string&foo=bar
real path=/
query=q=query+string&foo=bar
request_version=HTTP/1.0
```

```
SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

要使用变量的不同出现向查询串传入一个值序列，需要在调用 `urllencode()` 时将 `doseq` 设置为 `True`。

```
import urllib

query_args = { 'foo':['fool', 'foo2'] }
print 'Single :', urllib.urllencode(query_args)
print 'Sequence:', urllib.urllencode(query_args, doseq=True )
```

结果是一个查询串，同一个名称与多个值关联。

```
$ python urllib_urllencode_doseq.py

Single : foo=%5B%27fool%27%2C+%27foo2%27%5D
Sequence: foo=fool&foo=foo2
```

要对这个查询串解码，参见 `cgi` 模块的 `FieldStorage` 类。

查询参数中可能有一些特殊字符，在服务器端对 URL 解析时这些字符会带来问题，所以在传递到 `urllencode()` 时要对这些特殊字符“加引号”。要在本地对特殊字符加引号从而得到字符串的“安全”版本，可以直接使用 `quote()` 或 `quote_plus()` 函数。

```
import urllib

url = 'http://localhost:8080/~dhellmann/'
print 'urllencode() :', urllib.urllencode({'url':url})
print 'quote()      :', urllib.quote(url)
print 'quote_plus():', urllib.quote_plus(url)
```

`quote_plus()` 实现的加引号处理会更大程度地替换相应字符。


```
$ python urllib_quote.py
```

```
urlencode() : url=http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F
quote()      : http%3A//localhost%3A8080/%7Edhellmann/
quote_plus() : http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F
```

要完成加引号操作的逆过程，可以根据需要相应地使用 `unquote()` 或 `unquote_plus()`。

```
import urllib
```

```
print urllib.unquote('http%3A//localhost%3A8080/%7Edhellmann/')
print urllib.unquote_plus(
    'http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F'
)
```

编码值会转换回一个常规的 URL 字符串。

```
$ python urllib_unquote.py
```

```
http://localhost:8080/~dhellmann/
http://localhost:8080/~dhellmann/
```

12.3.3 路径与 URL

有些操作系统在本地文件和 URL 中使用不同的值分隔路径的不同部分。为了保证代码可移植，可以使用函数 `pathname2url()` 和 `url2pathname()` 来回转换。

注意：由于这些例子是为 Mac OS X 准备的，所以必须显式地导入函数的 Windows 版本。如果使用 `urllib` 导出的函数版本，会针对当前平台提供正确的默认值，所以大多数程序不需要显式导入。

```
import os
```

```
from urllib import pathname2url, url2pathname
```

```
print '== Default =='
path = '/a/b/c'
print 'Original:', path
print 'URL      :', pathname2url(path)
print 'Path     :', url2pathname('/d/e/f')
print
```

```
from nturl2path import pathname2url, url2pathname
```

```
print '== Windows, without drive letter =='
path = r'\a\b\c'
print 'Original:', path
print 'URL      :', pathname2url(path)
print 'Path     :', url2pathname('/d/e/f')
```



```

print

print '== Windows, with drive letter =='
path = r'C:\a\b\c'
print 'Original:', path
print 'URL      :', pathname2url(path)
print 'Path     :', url2pathname('/d/e/f')

```

这里有两个 Windows 例子，其路径前缀中分别包含和不包含驱动器字母。

```
$ python urllib_pathnames.py
```

```

== Default ==
Original: /a/b/c
URL      : /a/b/c
Path     : /d/e/f
== Windows, without drive letter ==
Original: \a\b\c
URL      : /a/b/c
Path     : \d\e\f

== Windows, with drive letter ==
Original: C:\a\b\c
URL      : //C:/a/b/c
Path     : \d\e\f

```

参见：

`urllib` (<http://docs.python.org/lib/module-urllib.html>) 这个模块的标准库文档。

`urllib2` (12.4 节) 处理基于 URL 服务的更新 API。

`urlparse` (12.1 节) 解析 URL 值来访问其组成部分。

12.4 urllib2——网络资源访问

作用：用于打开扩展 URL 的库，这些 URL 可以通过定义定制协议处理器来扩展。

Python 版本：2.1 及以后版本

`urllib2` 模块提供了一个更新的 API 来使用 URL 标识的 Internet 资源。可以由单个应用扩展来支持新的协议，或者可以对现有协议添加变化（如处理 HTTP 基本验证）。

12.4.1 HTTP GET

注意：这些例子的测试服务器在 `BaseHTTPServer_GET.py` 中（见 `BaseHTTPServer` 模块的相关例子）。在一个终端窗口启动服务器，然后在另一个终端窗口运行这些例子。

与 `urllib` 类似，HTTP GET 操作是 `urllib2` 最简单的用法。将 URL 传入 `urlopen()`，来得到远

程数据的一个“类文件”的句柄。

```
import urllib2

response = urllib2.urlopen('http://localhost:8080/')
print 'RESPONSE:', response
print 'URL      :', response.geturl()

headers = response.info()
print 'DATE      :', headers['date']
print 'HEADERS : '
print '-----'
print headers

data = response.read()
print 'LENGTH  :', len(data)
print 'DATA      : '
print '-----'
print data
```

这个示例服务器接收到来的值，并建立一个纯文本的格式化响应发回给客户。利用 `urlopen()` 的返回值，可以通过 `info()` 方法从 HTTP 服务器访问首部，还可以通过类似 `read()` 和 `readlines()` 等方法访问远程资源。

```
$ python urllib2_urlopen.py
```

```
RESPONSE: <addinfourl at 11940488 whose fp = <socket._fileobject
object at 0xb573f0>>
URL      : http://localhost:8080/
DATE     : Sun, 19 Jul 2009 14:01:31 GMT
HEADERS  :
-----
Server: BaseHTTP/0.3 Python/2.6.2
Date: Sun, 19 Jul 2009 14:01:31 GMT

LENGTH  : 349
DATA     :
-----
CLIENT VALUES:
client_address=('127.0.0.1', 55836) (localhost)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.3
```



```
sys_version=Python/2.6.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
accept-encoding=identity
connection=close
host=localhost:8080
user-agent=Python-urllib/2.6
```

urlopen() 返回的类文件对象是可迭代的。

```
import urllib2

response = urllib2.urlopen('http://localhost:8080/')
for line in response:
    print line.rstrip()
```

这个例子在打印输出之前先去除末尾的换行和回车。

```
$ python urllib2_urlopen_iterator.py
```

```
CLIENT VALUES:
client_address=('127.0.0.1', 55840) (localhost)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1
```

```
SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.6.2
protocol_version=HTTP/1.0
```

```
HEADERS RECEIVED:
accept-encoding=identity
connection=close
host=localhost:8080
user-agent=Python-urllib/2.6
```

12.4.2 参数编码

可以用 `urllib.urlencode()` 对参数编码，并追加到 URL，从而将参数传递到服务器。

```
import urllib
import urllib2

query_args = { 'q': 'query string', 'foo': 'bar' }
encoded_args = urllib.urlencode(query_args)
print 'Encoded:', encoded_args
```

```
url = 'http://localhost:8080/?' + encoded_args
print urllib2.urlopen(url).read()
```

示例输出返回的客户值列表包含有已编码的查询参数。

```
$ python urllib2_http_get_args.py
```

```
Encoded: q=query+string&foo=bar
CLIENT VALUES:
client_address=('127.0.0.1', 55849) (localhost)
command=GET
path=/?q=query+string&foo=bar
real path=/
query=q=query+string&foo=bar
request_version=HTTP/1.1
```

```
SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.6.2
protocol_version=HTTP/1.0
```

```
HEADERS RECEIVED:
accept-encoding=identity
connection=close
host=localhost:8080
user-agent=Python-urllib/2.6
```

12.4.3 HTTP POST

注意：这些例子的测试服务器在 `BaseHTTPServer_POST.py` 中（见 `BaseHTTPServer` 模块的相关例子）。在一个终端窗口中启动服务器，然后在另一个终端窗口中运行这些例子。

要使用 POST 而不是 GET 将表单编码（form-encoded）数据发送到远程服务器，需要将编码的查询参数作为数据传入 `urlopen()`。

```
import urllib
import urllib2

query_args = { 'q': 'query string', 'foo': 'bar' }
encoded_args = urllib.urlencode(query_args)
url = 'http://localhost:8080/'
print urllib2.urlopen(url, encoded_args).read()
```

服务器可以对表单数据解码，并按名称访问各个值。

```
$ python urllib2_urlopen_post.py
```

```
Client: ('127.0.0.1', 55943)
```

```
User-agent: Python-urllib/2.6
Path: /
Form data:
    q=query string
    foo=bar
```

12.4.4 增加发出首部

`urlopen()` 是一个便利函数，可以隐藏建立和处理请求的一些细节。通过直接使用 `Request` 实例可以提供更精确的控制。例如，可以向发出的请求增加定制首部，控制所返回数据的格式、指定本地缓存文档的版本，还可以告诉远程服务器与之通信的软件客户名。

如前例的输出所示，默认的用户-agent首部值包括常量 `Python-urllib`，后面是 Python 解释器版本。如果要创建一个应用访问其他人拥有的 Web 资源，最好在请求中包含真实的用户-agent 信息，这样比较礼貌，可以更容易地标识请求来源。通过使用一个定制代理，还允许使用一个 `robots.txt` 文件控制“爬虫”（参见 `robotparser` 模块）。

```
import urllib2

request = urllib2.Request('http://localhost:8080/')
request.add_header(
    'User-agent',
    'PyMOTW (http://www.doughellmann.com/PyMOTW/)',
)

response = urllib2.urlopen(request)
data = response.read()
print data
```

创建一个 `Request` 对象之后，打开请求之前使用 `add_header()` 设置用户-agent 值。输出的最后一行显示了这个定制值。

```
$ python urllib2_request_header.py

CLIENT VALUES:
client_address=('127.0.0.1', 55876) (localhost)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.6.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
```



```

accept-encoding=identity
connection=close
host=localhost:8080
user-agent=PyMOTW (http://www.doughellmann.com/PyMOTW/)

```

12.4.5 从请求提交表单数据

可以将发出的数据添加到 Request, 提交给服务器。

```

import urllib
import urllib2

query_args = { 'q':'query string', 'foo':'bar' }

request = urllib2.Request('http://localhost:8080/')
print 'Request method before data:', request.get_method()

request.add_data(urllib.urlencode(query_args))
print 'Request method after data:', request.get_method()
request.add_header(
    'User-agent',
    'PyMOTW (http://www.doughellmann.com/PyMOTW/)',
)

print
print 'OUTGOING DATA:'
print request.get_data()

print
print 'SERVER RESPONSE:'
print urllib2.urlopen(request).read()

```

添加数据之后, Request 使用的 HTTP 方法会自动从 GET 变为 POST。

```
$ python urllib2_request_post.py
```

```

Request method before data: GET
Request method after data : POST

```

```

OUTGOING DATA:
q=query+string&foo=bar

```

```

SERVER RESPONSE:
Client: ('127.0.0.1', 56044)
User-agent: PyMOTW (http://www.doughellmann.com/PyMOTW/)
Path: /
Form data:
  q=query string
  foo=bar

```



注意：尽管方法名为 `add_data()`，但其效果并不累计。每次调用都会替换原来的数据。

12.4.6 上传文件

与简单表单相比，要对文件编码以便上传，需要多做一些工作。要在请求体中构造一个完整的 MIME 消息，使服务器能够区分哪些是收到的表单域，哪些是上传的文件。

```
import itertools
import mimetools
import mimetypes
from cStringIO import StringIO
import urllib
import urllib2

class MultiPartForm(object):
    """Accumulate the data to be used when posting a form."""

    def __init__(self):
        self.form_fields = []
        self.files = []
        self.boundary = mimetools.choose_boundary()
        return

    def get_content_type(self):
        return 'multipart/form-data; boundary=%s' % self.boundary

    def add_field(self, name, value):
        """Add a simple field to the form data."""
        self.form_fields.append((name, value))
        return

    def add_file(self, fieldname, filename, fileHandle,
                 mimetype=None):
        """Add a file to be uploaded."""
        body = fileHandle.read()
        if mimetype is None:
            mimetype = (mimetypes.guess_type(filename)[0]
                        or
                        'application/octet-stream')
        self.files.append((fieldname, filename, mimetype, body))
        return

    def __str__(self):
        """Return a string representing the form data,
        including attached files.
```



```

"""
# Build a list of lists, each containing "lines" of the
# request. Each part is separated by a boundary string.
# Once the list is built, return a string where each
# line is separated by '\r\n'.
parts = []
part_boundary = '--' + self.boundary

# Add the form fields
parts.extend(
    [ part_boundary,
      'Content-Disposition: form-data; name="%s"' % name,
      '',
      value,
    ]
    for name, value in self.form_fields
)

# Add the files to upload
parts.extend([
    part_boundary,
    'Content-Disposition: file; name="%s"; filename="%s"' % \
      (field_name, filename),
    'Content-Type: %s' % content_type,
    '',
    body,
  ])
  for field_name, filename, content_type, body in self.files
)

# Flatten the list and add closing boundary marker, and
# then return CR+LF separated data
flattened = list(itertools.chain(*parts))
flattened.append('--' + self.boundary + '--')
flattened.append('')
return '\r\n'.join(flattened)

if __name__ == '__main__':
    # Create the form with simple fields
    form = MultiPartForm()
    form.add_field('firstname', 'Doug')
    form.add_field('lastname', 'Hellmann')

    # Add a fake file
    form.add_file(
        'biography', 'bio.txt',
        fileHandle=StringIO('Python developer and blogger.'))

```

```

# Build the request
request = urllib2.Request('http://localhost:8080/')
request.add_header(
    'User-agent',
    'PyMOTW (http://www.doughellmann.com/PyMOTW/)')
body = str(form)
request.add_header('Content-type', form.get_content_type())
request.add_header('Content-length', len(body))
request.add_data(body)

print
print 'OUTGOING DATA:'
print request.get_data()

print
print 'SERVER RESPONSE:'
print urllib2.urlopen(request).read()

```

MultiPartForm 类可以把一个任意的表单表示为带附加文件的多部分 MIME 消息。

```
$ python urllib2_upload_files.py
```

```

OUTGOING DATA:
--192.168.1.17.527.30074.1248020372.206.1
Content-Disposition: form-data; name="firstname"
Doug
--192.168.1.17.527.30074.1248020372.206.1
Content-Disposition: form-data; name="lastname"

Hellmann
--192.168.1.17.527.30074.1248020372.206.1
Content-Disposition: file; name="biography"; filename="bio.txt"
Content-Type: text/plain

```

```

Python developer and blogger.
--192.168.1.17.527.30074.1248020372.206.1--

```

```

SERVER RESPONSE:
Client: ('127.0.0.1', 57126)
User-agent: PyMOTW (http://www.doughellmann.com/PyMOTW/)
Path: /
Form data:
  lastname=Hellmann
  Uploaded biography as "bio.txt" (29 bytes)
  firstname=Doug

```



12.4.7 创建定制协议处理器

urllib2 提供了对 HTTP(S)、FTP 和本地文件访问的内置支持。要增加对其他 URL 类型的支持，需要注册另外的协议处理器。例如，要支持指向远程 NFS 服务器上任意文件的 URL，而不需要用户在访问文件之前先装载 (mount) 这个路径，可以创建一个由 BaseHandler 派生的类，并包括一个 nfs_open() 方法。

协议特定的 open() 方法 (如 nfs_open()) 会得到一个参数，即 Request 实例，它要返回一个对象，这个对象包括一个 read() 方法，可以用来读取数据，另外包括一个 info() 方法返回响应首部，还有一个 geturl() 方法返回所读取文件的具体 URL。要得到这个结果，一种简单的方法是创建一个 urllib.addurlinfo 实例，在构造函数中传入首部、URL 和打开的文件句柄。

```
import mimetypes
import os
import tempfile
import urllib
import urllib2

class NFSFile(file):
    def __init__(self, tempdir, filename):
        self.tempdir = tempdir
        file.__init__(self, filename, 'rb')
    def close(self):
        print 'NFSFile:'
        print '  unmounting %s' % os.path.basename(self.tempdir)
        print '  when %s is closed' % os.path.basename(self.name)
        return file.close(self)

class FauxNFSHandler(urllib2.BaseHandler):

    def __init__(self, tempdir):
        self.tempdir = tempdir

    def nfs_open(self, req):
        url = req.get_selector()
        directory_name, file_name = os.path.split(url)
        server_name = req.get_host()
        print 'FauxNFSHandler simulating mount:'
        print '  Remote path: %s' % directory_name
        print '  Server      : %s' % server_name
        print '  Local path : %s' % os.path.basename(tempdir)
        print '  Filename   : %s' % file_name
        local_file = os.path.join(tempdir, file_name)
        fp = NFSFile(tempdir, local_file)
        content_type = (mimetypes.guess_type(file_name)[0]
                        or
```

```

        'application/octet-stream'
    )
    stats = os.stat(local_file)
    size = stats.st_size
    headers = { 'Content-type': content_type,
                'Content-length': size,
                }
    return urllib.addinfourl(fp, headers, req.get_full_url())

if __name__ == '__main__':
    tempdir = tempfile.mkdtemp()
    try:
        # Populate the temporary file for the simulation
        with open(os.path.join(tempdir, 'file.txt'), 'wt') as f:
            f.write('Contents of file.txt')

        # Construct an opener with our NFS handler
        # and register it as the default opener.
        opener = urllib2.build_opener(FauxNFSHandler(tempdir))
        urllib2.install_opener(opener)

        # Open the file through a URL.
        response = urllib2.urlopen(
            'nfs://remote_server/path/to/the/file.txt'
        )
        print
        print 'READ CONTENTS:', response.read()
        print 'URL          :', response.geturl()
        print 'HEADERS:'
        for name, value in sorted(response.info().items()):
            print ' %15s = %s' % (name, value)
        response.close()
    finally:
        os.remove(os.path.join(tempdir, 'file.txt'))
        os.removedirs(tempdir)

```

FauxNFSHandler 和 NFSFile 类打印了一些消息，来展示真正的实现会在哪里增加装载 (mount) 和卸载 (unmount) 调用。由于这只是一个模拟，FauxNFSHandler 只是简单地提供了一个临时目录名，它将在这个目录中查找所有文件。

```
$ python urllib2_nfs_handler.py
```

```

FauxNFSHandler simulating mount:
Remote path: /path/to/the
Server      : remote_server
Local path  : tmpoqqoAV
Filename    : file.txt

```

```

READ CONTENTS: Contents of file.txt
URL           : nfs://remote_server/path/to/the/file.txt
HEADERS:
  Content-length = 20
  Content-type   = text/plain
NFSFile:
  unmounting tmpoqgoAV
  when file.txt is closed

```

参见:

`urllib2` (<http://docs.python.org/library/urllib2.html>) 这个模块的标准库文档。

`urllib` (12.3 节) 原 URL 处理库。

`urlparse` (12.1 节) 处理 URL 串本身。

`urllib2` – The Missing Manual (www.voidspace.org.uk/python/articles/urllib2.shtml) Michael Foord 撰写的关于使用 `urllib2` 的文章。

Upload Scripts (www.voidspace.org.uk/python/cgi.shtml#upload) Michael Foord 提供的示例脚本, 展示了如何使用 HTTP 上传一个文件, 然后在服务器接收数据。

HTTP client to POST using multipart/form-data (<http://code.activestate.com/recipes/146306>) Python 实用手册, 显示了如何对数据编码, 以及通过 HTTP 提交数据 (包括文件)。

Form content types (www.w3.org/TR/REC-html40/interact/forms.html#h-17.13.4) 通过 HTTP 表单提交文件或大量数据的 W3C 规范。

`Mimetypes` 将文件名映射到 `mimetype`。

`Mimertools` 解析 MIME 消息的工具。

12.5 Base64——用 ASCII 编码二进制数据

作用: `base64` 模块包含一些函数, 可以将二进制数据转换为适合使用纯文本协议传输的 ASCII 子集。

Python 版本: 1.4 及以后版本

`Base64`、`Base32` 和 `Base16` 编码将 8 位字节分别转换为每字节有 6、5 或 4 位有用数据的值, 这就允许将非 ASCII 字节编码为 ASCII 字符, 以便通过需要纯 ASCII 的协议进行传输, 如 SMTP。`base`(进制) 值对应于各编码中使用的字母表长度。这些原始编码还有一些“URL 安全”(URL-safe) 的变种, 使用的字母表稍有不同。

12.5.1 Base64 编码

以下是对文本编码的一个基本例子。

```

import base64
import textwrap

```

```

# Load this source file and strip the header.
with open(__file__, 'rt') as input:
    raw = input.read()
    initial_data = raw.split('#end_pymotw_header')[1]

encoded_data = base64.b64encode(initial_data)

num_initial = len(initial_data)

# There will never be more than 2 padding bytes.
padding = 3 - (num_initial % 3)

print '%d bytes before encoding' % num_initial
print 'Expect %d padding bytes' % padding
print '%d bytes after encoding' % len(encoded_data)
print
print encoded_data

```

输出显示出原始源文件的 168 字节经编码后扩展为 224 字节。

注意：由库生成的编码数据中并没有回车，不过这里的输出中人工添加了换行符，从而能更美观地在页面上显示。

```
$ python base64_b64encode.py
```

```

168 bytes before encoding
Expect 3 padding bytes
224 bytes after encoding

```

```

CgppbXBvcnQgYmFzZTY0CmltcG9ydCB0ZXh0d3JhcAoKIyBMb2FkIHRoaXMgc291c
mNlIGZpbGUgYW5kIHN0cmVzZSBoZWZkZXIuCndpdGggb3BlbihfX2ZpbGVfXy
wgJ3J0JykgYXMgaW5wdXQ6CiAgICByYXcgPSBpbmBldC5yZWZkKCKKICAgIGluaXR
pYWxfZGF0YSA9IHJhdy5zcGxpCgn

```

12.5.2 Base64 解码

`b64decode()` 将编码的串转换回原来的形式，它取 4 个字节，利用一个查找表将这 4 个字节转换回原来的 3 个字节。

```

import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b64encode(original_string)

```

```
print 'Encoded :', encoded_string

decoded_string = base64.b64decode(encoded_string)
print 'Decoded :', decoded_string
```

编码过程中，会查看输入中的各个 24 位序列（3 个字节），然后将这 24 位编码为输出中的 4 个字节。输出末尾的等号作为填充插入，因为在这个例子中，原始串中的位数不能被 24 整除。

```
$ python base64_b64decode.py

Original: This is the data, in the clear.
Encoded : VGhpcyBpcyB0aGUgZGF0YSwgaW4gdGhlIGNsZWZyLg==
Decoded : This is the data, in the clear.
```

12.5.3 URL 安全的变种

因为默认的 Base64 字母表可能使用 + 和 /，这两个字符在 URL 中会用到，所以通常很有必要使用一个候选编码，来替换这些字符。

```
import base64

encodes_with_pluses = chr(251) + chr(239)
encodes_with_slashes = chr(255) * 2

for original in [ encodes_with_pluses, encodes_with_slashes ]:
    print 'Original          :', repr(original)
    print 'Standard encoding:', base64.standard_b64encode(original)
    print 'URL-safe encoding:', base64.urlsafe_b64encode(original)
    print
```

+ 替换为 -, / 替换为下划线 ()。除此之外，字母表是一样的。

```
$ python base64_urlsafe.py

Original          : '\xfb\xef'
Standard encoding: ++8=
URL-safe encoding: --8=

Original          : '\xff\xff'
Standard encoding: //8=
URL-safe encoding: __8=
```

12.5.4 其他编码

除了 Base64，这个模块还提供了一些函数来处理 Base32 和 Base16（十六进制）编码数据。

```
import base64

original_string = 'This is the data, in the clear.'
```

```

print 'Original:', original_string

encoded_string = base64.b32encode(original_string)
print 'Encoded :', encoded_string

decoded_string = base64.b32decode(encoded_string)
print 'Decoded :', decoded_string

```

Base32 字母表包括 ASCII 集中的 26 个大写字母以及数字 2~7。

```
$ python base64_base32.py
```

```

Original: This is the data, in the clear.
Encoded : KRUGS4ZANFZSA5DIMUQGIYLUMEWCA2LOEB2GQZJAMNWGKYLSFY=====
Decoded : This is the data, in the clear.

```

Base16 函数处理十六进制字母表。

```

import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b16encode(original_string)
print 'Encoded :', encoded_string
decoded_string = base64.b16decode(encoded_string)
print 'Decoded :', decoded_string

```

每次编码位数下降时，采用编码格式的输出就会占用更多空间。

```
$ python base64_base16.py
```

```

Original: This is the data, in the clear.
Encoded : 546869732069732074686520646174612C20696E2074686520636C6561
722E
Decoded : This is the data, in the clear.

```

参见：

base64 (<http://docs.python.org/library/base64.html>) 这个模块的标准库文档。

RFC 3548 (<http://tools.ietf.org/html/rfc3548.html>) Base16、Base32 和 Base64 数据编码。

12.6 robotparser——网络蜘蛛访问控制

作用：解析用于控制网络蜘蛛的 robots.txt 文件。

Python 版本：2.1.3 及以后版本

robotparser 为 robots.txt 文件格式实现一个解析器，包括一个函数来检查给定的 user-agent 是否可以访问一个资源。这个模块可以用于需要抑制或限制的合法蜘蛛或其他爬虫应用。

12.6.1 robots.txt

robots.txt 文件格式是一个基于文本的简单访问控制系统，用于自动访问 Web 资源的计算机程序（如“蜘蛛”、“爬虫”等等）。这个文件由记录构成，各记录会指定程序的 user-agent 标识符，后面是代理不能访问的一个 URL（或 URL 前缀）列表。

以下是 <http://www.doughellmann.com/> 的 robots.txt 文件。

```
User-agent: *
Disallow: /admin/
Disallow: /downloads/
Disallow: /media/
Disallow: /static/
Disallow: /codehosting/
```

这会阻止访问网站中某些计算代价昂贵的部分，如果搜索引擎试图索引这些部分，可能会使服务器负载过重。要得到 robots.txt 更为完整的示例集，可以参考 Web Robots 页面（见本节后面的参见列表）。

12.6.2 测试访问权限

基于之前提供的数据，一个简单的爬虫应用可以使用 `RobotFileParser.can_fetch()` 测试是否允许下载一个页面。

```
import robotparser
import urlparse

AGENT_NAME = 'PyMOTW'
URL_BASE = 'http://www.doughellmann.com/'
parser = robotparser.RobotFileParser()
parser.set_url(urlparse.urljoin(URL_BASE, 'robots.txt'))
parser.read()

PATHS = [
    '/',
    '/PyMOTW/',
    '/admin/',
    '/downloads/PyMOTW-1.92.tar.gz',
]

for path in PATHS:
    print '%6s : %s' % (parser.can_fetch(AGENT_NAME, path), path)
    url = urlparse.urljoin(URL_BASE, path)
    print '%6s : %s' % (parser.can_fetch(AGENT_NAME, url), url)
    print
```

`can_fetch()` 的 URL 参数可以是一个相对于网站根目录的相对路径，也可以是一个完全 URL。

```
$ python robotparser_simple.py

True : /
True : http://www.doughellmann.com/
True : /PyMOTW/
True : http://www.doughellmann.com/PyMOTW/

False : /admin/
False : http://www.doughellmann.com/admin/

False : /downloads/PyMOTW-1.92.tar.gz
False : http://www.doughellmann.com/downloads/PyMOTW-1.92.tar.gz
```

12.6.3 长久蜘蛛

如果一个应用需要花很长时间来处理它下载的资源，或者受到抑制，需要在多次下载之间暂停，这样的应用应当以其已下载内容的寿命为根据，定期检查新的 robots.txt 文件。这个寿命并不是自动管理的，不过模块提供了一些简便方法，利用这些方法可以更容易地跟踪文件的寿命。

```
import robotparser
import time
import urlparse

AGENT_NAME = 'PyMOTW'
parser = robotparser.RobotFileParser()
# Using the local copy
parser.set_url('robots.txt')
parser.read()
parser.modified()

PATHS = [
    '/',
    '/PyMOTW/',
    '/admin/',
    '/downloads/PyMOTW-1.92.tar.gz',
]

for path in PATHS:
    age = int(time.time() - parser.mtime())
    print 'age:', age,
    if age > 1:
        print 'rereading robots.txt'
        parser.read()
        parser.modified()
    else:
        print
    print '%6s : %s' % (parser.can_fetch(AGENT_NAME, path), path)
```

```
# Simulate a delay in processing
time.sleep(1)
print
```

如果已下载的文件寿命超过了 1 秒，这个极端例子就会下载一个新的 robots.txt 文件。

```
$ python robotparser_longlived.py
```

```
age: 0
True : /
```

```
age: 1
True : /PyMOTW/
```

```
age: 2 rereading robots.txt
False : /admin/
```

```
age: 1
False : /downloads/PyMOTW-1.92.tar.gz
```

作为一个更好的长久应用，在下载整个文件之前可能会请求文件的修改时间。另一方面，robots.txt 文件通常很小，所以再次获取整个文档的开销并不昂贵。

参见：

robotparser (<http://docs.python.org/library/robotparser.html>) 这个模块的标准库文档。

The Web Robots Page (www.robotstxt.org/orig.html) 对 robots.txt 格式的描述。

12.7 Cookie——HTTP Cookie

作用：Cookie 模块定义一些类来解析和创建 HTTP cookie 首部。

Python 版本：2.1 及以后版本

Cookie 模块为大多数符合 RFC 2109 的 cookie 实现一个解析器。这个实现没有标准那么严格，因为 MSIE 3.0x 并不支持整个标准。

12.7.1 创建和设置 Cookie

可以用 Cookie 为基于浏览器的应用实现状态管理，因此，Cookie 通常由服务器设置，并由客户存储和返回。下面是创建一个 cookie 的最简单的例子。

```
import Cookie

c = Cookie.SimpleCookie()
c['mycookie'] = 'cookie_value'
print c
```

输出是一个合法的 Set-Cookie 首部，可以作为 HTTP 响应的一部分传递到客户。

```
$ python Cookie_setheaders.py

Set-Cookie: mycookie=cookie_value
```

12.7.2 Morsel

还可以控制 cookie 的其他方面，如到期时间、路径和域。实际上，cookie 的所有 RFC 属性都可以通过表示 cookie 值的 Morsel 对象来管理。

```
import Cookie
import datetime

def show_cookie(c):
    print c
    for key, morsel in c.iteritems():
        print
        print 'key =', morsel.key
        print ' value =', morsel.value
        print ' coded_value =', morsel.coded_value
        for name in morsel.keys():
            if morsel[name]:
                print ' %s = %s' % (name, morsel[name])

c = Cookie.SimpleCookie()

# A cookie with a value that has to be encoded to fit into the header
c['encoded_value_cookie'] = '"cookie_value"'
c['encoded_value_cookie']['comment'] = 'Value has escaped quotes'

# A cookie that only applies to part of a site
c['restricted_cookie'] = 'cookie_value'
c['restricted_cookie']['path'] = '/sub/path'
c['restricted_cookie']['domain'] = 'PyMOTW'
c['restricted_cookie']['secure'] = True

# A cookie that expires in 5 minutes
c['with_max_age'] = 'expires in 5 minutes'
c['with_max_age']['max-age'] = 300 # seconds

# A cookie that expires at a specific time
c['expires_at_time'] = 'cookie_value'
time_to_live = datetime.timedelta(hours=1)
expires = datetime.datetime(2009, 2, 14, 18, 30, 14) + time_to_live

# Date format: Wdy, DD-Mon-YY HH:MM:SS GMT
expires_at_time = expires.strftime('%a, %d %b %Y %H:%M:%S')
c['expires_at_time']['expires'] = expires_at_time
```

```
show_cookie(c)
```

这个例子使用两个不同的方法设置所存储 cookie 的到期时间。其中一个将 max-age 设置为秒数，另一个将 expires 设置为日期时间，达到这个日期时间就会丢弃这个 cookie。

```
$ python Cookie_Morsel.py
```

```
Set-Cookie: encoded_value_cookie="\cookie_value\"; Comment=Value h
as escaped quotes
Set-Cookie: expires_at_time=cookie_value; expires=Sat, 14 Feb 2009 1
9:30:14
Set-Cookie: restricted_cookie=cookie_value; Domain=PyMOTW; Path=/sub
/path; secure
Set-Cookie: with_max_age="expires in 5 minutes"; Max-Age=300
key = restricted_cookie
    value = cookie_value
    coded_value = cookie_value
    domain = PyMOTW
    secure = True
    path = /sub/path

key = with_max_age
    value = expires in 5 minutes
    coded_value = "expires in 5 minutes"
    max-age = 300

key = encoded_value_cookie
    value = "cookie_value"
    coded_value = "\cookie_value\"
    comment = Value has escaped quotes

key = expires_at_time
    value = cookie_value
    coded_value = cookie_value
    expires = Sat, 14 Feb 2009 19:30:14
```

Cookie 和 Morsel 对象与字典类似。Morsel 响应一个固定的键集：

- expires
- path
- comment
- domain
- max-age
- secure
- version

Cookie 实例的键是所存储的各个 cookie 的名称。这个信息也可以从 Morsel 的键属性得到。

12.7.3 编码值

cookie 首部要求对值编码，才能正确地解析。

```
import Cookie

c = Cookie.SimpleCookie()
c['integer'] = 5
c['string_with_quotes'] = 'He said, "Hello, World!"'

for name in ['integer', 'string_with_quotes']:
    print c[name].key
    print ' %s' % c[name]
    print ' value=%r' % c[name].value
    print ' coded_value=%r' % c[name].coded_value
    print
```

Morsel.value 是 cookie 的解码值，而 Morsel.coded_value 表示则用来将值传输到客户。这两个值都是串。如果保存到 cookie 的值不是串将会自动转换。

```
$ python Cookie_coded_value.py

integer
Set-Cookie: integer=5
value='5'
coded_value='5'

string_with_quotes
Set-Cookie: string_with_quotes="He said, \"Hello, World!\""
value='He said, "Hello, World!"'
coded_value='\"He said, \\\"Hello, World!\\\"\"'
```

12.7.4 接收和解析 Cookie 首部

一旦客户接收到 Set-Cookie 首部，在后续请求中它会使用一个 Cookie 首部把这些 cookie 返回到服务器。到来的 Cookie 首部串可能包含多个 cookie 值，由分号分隔(;)。

```
Cookie: integer=5; string_with_quotes="He said, \"Hello, World!\""
```

取决于 Web 服务器和框架，可以直接从首部或 HTTP_COOKIE 环境变量得到 cookie。

```
import Cookie

HTTP_COOKIE = '; '.join([
    r'integer=5',
    r'string_with_quotes="He said, \"Hello, World!\"",
])

print 'From constructor:'
c = Cookie.SimpleCookie(HTTP_COOKIE)
```

```

print c

print
print 'From load():'
c = Cookie.SimpleCookie()
c.load(HTTP_COOKIE)
print c

```

要对其解码，实例化时可以将串（但不包括首部前缀）传递到 SimpleCookie，或者使用 load() 方法。

```
$ python Cookie_parse.py
```

```

From constructor:
Set-Cookie: integer=5
Set-Cookie: string_with_quotes="He said, \"Hello, World!\""

From load():
Set-Cookie: integer=5
Set-Cookie: string_with_quotes="He said, \"Hello, World!\""

```

12.7.5 候选输出格式

除了使用 Set-Cookie 首部外，服务器还可以提供 JavaScript，向客户添加 cookie。SimpleCookie 和 Morsel 通过 js_output() 方法来提供 JavaScript 输出。

```

import Cookie

c = Cookie.SimpleCookie()
c['mycookie'] = 'cookie_value'
c['another_cookie'] = 'second value'
print c.js_output()

```

结果是一个完整的 script 标记，其中包含设置 cookie 的语句。

```

$ python Cookie_js_output.py

<script type="text/javascript">
<!-- begin hiding
document.cookie = "another_cookie=\"second value\"";
// end hiding -->
</script>

<script type="text/javascript">
<!-- begin hiding
document.cookie = "mycookie=cookie_value";
// end hiding -->
</script>

```



12.7.6 废弃的类

所有这些例子都使用了 SimpleCookie。Cookie 模块还提供了另外两个类，SerialCookie 和 SmartCookie。SerialCookie 可以处理任何可 pickle 的值。SmartCookie 能确定一个值是否需要解除 pickle，或者这是否是一个简单值。

警告： 由于这两个类都使用 pickle，它们存在潜在的安全漏洞，最好不要使用。更安全的做法是在服务器上存储状态，并为客户提供一个会话密钥。

参见：

Cookie (<http://docs.python.org/library/cookie.html>) 这个模块的标准库文档。

cookielib cookielib 模块在客户端处理 cookie。

RFC 2109 (<http://tools.ietf.org/html/rfc2109.html>) HTTP 状态管理机制。

12.8 uuid——全局惟一标识符

作用：uuid 模块实现了全局惟一标识符，如 RFC 4122 所述。

Python 版本：2.5 及以后版本

RFC 4122 定义了一个系统，可以为资源创建全局惟一标识符（Universally Unique Identifier），这里采用一种不需要集中注册机的方式。UUID 值为 128 位，正如参考指南所述，“UUID 可以保证跨空间和时间的惟一性”。它们对于为文档、主机、应用客户以及其他需要惟一值的情况生成标识符很有用。RFC 特别强调创建一个统一资源名（Uniform Resource Name）命名空间，并涵盖了 3 个主要算法。

- 使用 IEEE 802 MAC 地址作为惟一性来源
- 使用伪随机数
- 使用公开的串并结合密码散列

在上述所有情况下，种子值都要与系统时钟结合，如果向后设置时钟，则要用一个时钟序列值维护惟一性。

12.8.1 UUID 1——IEEE 802 MAC 地址

UUID 1 值使用主机的 MAC 地址计算。uuid 模块使用 getnode() 来获取当前系统的 MAC 值。

```
import uuid
```

```
print hex(uuid.getnode())
```

如果一个系统有多个网卡，相应的有多个 MAC 地址，可能返回其中任意一个值。

```
$ python uuid_getnode.py
```

```
0x1e5274040e
```


要为一个主机生成一个 UUID，由其 MAC 地址标识，需要使用 `uuid1()` 函数。节点标识符参数是可选的；如果未设置这个域，就会使用 `getnode()` 返回的值。

```
import uuid

u = uuid.uuid1()

print u
print type(u)
print 'bytes      :', repr(u.bytes)
print 'hex        :', u.hex
print 'int         :', u.int
print 'urn         :', u.urn
print 'variant     :', u.variant
print 'version     :', u.version
print 'fields      :', u.fields
print '\ttime_low      : ', u.time_low
print '\ttime_mid      : ', u.time_mid
print '\ttime_hi_version : ', u.time_hi_version
print '\tclock_seq_hi_variant: ', u.clock_seq_hi_variant
print '\tclock_seq_low  : ', u.clock_seq_low
print '\tnode          : ', u.node
print '\ttime           : ', u.time
print '\tclock_seq      : ', u.clock_seq
```

对于返回的 UUID 对象，它的各个部分可以通过只读的实例属性访问。有些属性，如 `hex`、`int` 和 `urn`，则是 UUID 值的不同表示。

```
$ python uuid_uuid1.py
```

```
c7887eee-ea6a-11df-a6cf-001e5274040e
<class 'uuid.UUID'>
bytes      : '\xc7\x88~\xee\xej\x11\xdf\xa6\xcf\x00\x1eRt\x04\x0e'
hex        : c7887eeeee6a11dfa6cf001e5274040e
int         : 265225098046419456611671377169708483598
urn         : urn:uuid:c7887eee-ea6a-11df-a6cf-001e5274040e
variant     : specified in RFC 4122
version     : 1
fields      : (3347611374L, 60010L, 4575L, 166L, 207L, 130232353806L)
time_low    : 3347611374
time_mid    : 60010
time_hi_version : 4575
clock_seq_hi_variant: 166
clock_seq_low  : 207
node         : 130232353806
time         : 135084258179448558
clock_seq    : 9935
```

由于有时间分量 (time)，每次调用 `uuid1()` 都会返回一个新值。

```
import uuid

for i in xrange(3):
    print uuid.uuid1()
```

在这个输出中，只有时间分量（串的开始位置）有变化。

```
$ python uuid_uuid1_repeat.py
```

```
c794da9c-ea6a-11df-9382-001e5274040e
c797121c-ea6a-11df-9e67-001e5274040e
c79713a1-ea6a-11df-ac7d-001e5274040e
```

由于每个计算机有一个不同的 MAC 地址，在不同系统上运行这个示例程序会产生完全不同的值。这个例子传递不同的节点 id 来模拟在不同主机上运行。

```
import uuid

for node in [ 0x1ec200d9e0, 0x1e5274040e ]:
    print uuid.uuid1(node), hex(node)
```

除了时间值不同外，UUID 末尾的节点标识符也有变化。

```
$ python uuid_uuid1_othermac.py
```

```
c7a313a8-ea6a-11df-a228-001ec200d9e0 0x1ec200d9e0
c7a3f751-ea6a-11df-988b-001e5274040e 0x1e5274040e
```

12.8.2 UUID 3 和 5——基于名字的值

有些情况下可能需要根据名字创建 UUID 值，而不是根据随机值或基于时间的值来创建。UUID 3 和 5 规范使用密码散列值（分别使用 MD5 或 SHA1），将特定于命名空间的种子值与名字相结合。有一些公开的命名空间，由预定义的 UUID 值标识，分别用于处理 DNS、URL、ISO OID 和 X.500 识别名 (Distinguished Names)。通过生成和保存 UUID 值，还可以定义新的特定于应用的命名空间。

```
import uuid

hostnames = ['www.doughellmann.com', 'blog.doughellmann.com']

for name in hostnames:
    print name
    print ' MD5      :', uuid.uuid3(uuid.NAMESPACE_DNS, name)
    print ' SHA-1   :', uuid.uuid5(uuid.NAMESPACE_DNS, name)
    print
```

要从一个 DNS 名创建 UUID，将 `uuid.NAMESPACE_DNS` 作为命名空间参数传入 `uuid3()` 或 `uuid5()`。

```
$ python uuid_uuid3_uuid5.py
```

```
www.doughellmann.com
```

```
MD5      : bcd02e22-68f0-3046-a512-327cca9def8f
SHA-1    : e3329b12-30b7-57c4-8117-c2cd34a87ce9
```

```
blog.doughellmann.com
```

```
MD5      : 9bdabfce-dfd6-37ab-8a3f-7f7293bcf111
SHA-1    : fa829736-7ef8-5239-9906-b4775a5abacb
```

一个命名空间中给定名的 UUID 值总是相同的，而不论在何时或何地计算。

```
import uuid
```

```
namespace_types = sorted(n
                           for n in dir(uuid)
                           if n.startswith('NAMESPACE_')
                           )
name = 'www.doughellmann.com'
```

```
for namespace_type in namespace_types:
    print namespace_type
    namespace_uuid = getattr(uuid, namespace_type)
    print ' ', uuid.uuid3(namespace_uuid, name)
    print ' ', uuid.uuid3(namespace_uuid, name)
    print
```

但是命名空间中相同名的 UUID 值则是不同的。

```
$ python uuid_uuid3_repeat.py
```

```
NAMESPACE_DNS
```

```
bcd02e22-68f0-3046-a512-327cca9def8f
bcd02e22-68f0-3046-a512-327cca9def8f
```

```
NAMESPACE_OID
```

```
e7043ac1-4382-3c45-8271-d5c083e41723
e7043ac1-4382-3c45-8271-d5c083e41723
```

```
NAMESPACE_URL
```

```
5d0fdaa9-eafd-365e-b4d7-652500dd1208
5d0fdaa9-eafd-365e-b4d7-652500dd1208
```

```
NAMESPACE_X500
```

```
4a54d6e7-ce68-37fb-b0ba-09acc87cabb7
4a54d6e7-ce68-37fb-b0ba-09acc87cabb7
```



12.8.3 UUID 4——随机值

有时，基于主机和基于命名空间的 UUID 值“差别还不够大”。例如，如果 UUID 要用作散列键，则需要有区分度更大、更随机的值序列来避免散列表中出現冲突。通过让值有更少的共同数字，也能更容易地在日志文件中查找这些值。为了增加 UUID 的区分度，可以使用 `uuid4()` 用随机的输入值来生成。

```
import uuid

for i in xrange(3):
    print uuid.uuid4()
```

随机性的来源取决于导入 `uuid` 时哪些 C 库可用。如果可以加载 `libuuid`（或 `uuid.dll`），而且其中包含一个生成随机值的函数，则使用这个库。否则，使用 `os.urandom()` 或 `random` 模块。

```
$ python uuid_uuid4.py

b2637198-4629-44c2-8b9b-07a6ff601a89
d1b850c6-f842-4a25-a993-6d6160dda761
50fb5234-abce-40b8-b034-ba3637dad6fc
```

12.8.4 处理 UUID 对象

除了生成新的 UUID 值，还可以解析标准格式的串来创建 UUID 对象，使比较和排序操作的处理更为容易。

```
import uuid

def show(msg, l):
    print msg
    for v in l:
        print ' ', v
    print

input_values = [
    'urn:uuid:f2f84497-b3bf-493a-bba9-7c68e6def80b',
    '{417a5ebb-01f7-4ed5-aeac-3d56cd5037b0}',
    '2115773a-5bf1-11dd-ab48-001ec200d9e0',
]

show('input_values', input_values)

uuids = [ uuid.UUID(s) for s in input_values ]
show('converted to uuids', uuids)

uuids.sort()
show('sorted', uuids)
```

将外围大括号从输入中去除，另外将短横线(-)也去除。如果串有一个包含 urn: 或 uuid: 的前缀，也会将其删除。剩下的文本必然是十六进制数构成的串，然后再将它解释为一个 UUID 值。

```
$ python uuid_uuid_objects.py

input_values
urn:uuid:f2f84497-b3bf-493a-bba9-7c68e6def80b
{417a5ebb-01f7-4ed5-aeac-3d56cd5037b0}
2115773a-5bf1-11dd-ab48-001ec200d9e0

converted to uuids
f2f84497-b3bf-493a-bba9-7c68e6def80b
417a5ebb-01f7-4ed5-aeac-3d56cd5037b0
2115773a-5bf1-11dd-ab48-001ec200d9e0

sorted
2115773a-5bf1-11dd-ab48-001ec200d9e0
417a5ebb-01f7-4ed5-aeac-3d56cd5037b0
f2f84497-b3bf-493a-bba9-7c68e6def80b
```

参见：

uuid (<http://docs.python.org/lib/module-uuid.html>) 这个模块的标准库文档。

RFC 4122 (<http://tools.ietf.org/html/rfc4122.html>) 全局唯一标识符 (UUID) URN 命名空间。

12.9 json——JavaScript 对象记法

作用：将 Python 对象编码为 JSON 串，以及将 JSON 串解码为 Python 对象。

Python 版本：2.6 及以后版本

json 模块提供了一个与 pickle 类似的 API，可以将内存中的 Python 对象转换为一个序列化表示，称为 JavaScript 对象记法 (JavaScript Object Notation, JSON)。不同于 pickle，JSON 有一个优点，它有多种语言的实现（特别是 JavaScript）。JSON 最广泛地应用于 AJAX 应用中 Web 服务器和客户之间的通信，不过也可以用于满足其他应用间的通信需求。

12.9.1 编码和解码简单数据类型

默认情况下，编码器支持 Python 的一些内置类型 (string、unicode、int、float、list、tuple 和 dict)。

```
import json

data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0 } ]
print 'DATA:', repr(data)

data_string = json.dumps(data)
print 'JSON:', data_string
```

对值编码时，表面上类似于 Python 的 `repr()` 输出。

```
$ python json_simple_types.py
```

```
DATA: [{'a': 'A', 'c': 3.0, 'b': (2, 4)}]
JSON: [{"a": "A", "c": 3.0, "b": [2, 4]}]
```

编码然后再重新解码时，可能不会得到完全相同的对象类型。

```
import json

data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0 } ]
print 'DATA :', data

data_string = json.dumps(data)
print 'ENCODED:', data_string

decoded = json.loads(data_string)
print 'DECODED:', decoded

print 'ORIGINAL:', type(data[0]['b'])
print 'DECODED :', type(decoded[0]['b'])
```

具体地，字符串会转换为 `unicode` 对象，而元组会变成列表。

```
$ python json_simple_types_decode.py

DATA : [{'a': 'A', 'c': 3.0, 'b': (2, 4)}]
ENCODED: [{"a": "A", "c": 3.0, "b": [2, 4]}]
DECODED: [{'a': 'A', 'c': 3.0, 'b': [2, 4]}]
ORIGINAL: <type 'tuple'>
DECODED : <type 'list'>
```

12.9.2 优质输出和紧凑输出

JSON 优于 `pickle` 的另一个好处是，其结果是人类可读的。`dumps()` 函数接受多个参数，可以使输出更美观。例如，`sort_keys` 标志会告诉编码器按有序顺序而不是随机顺序输出字典的键。

```
import json

data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0 } ]
print 'DATA:', repr(data)

unsorted = json.dumps(data)
print 'JSON:', json.dumps(data)
print 'SORT:', json.dumps(data, sort_keys=True)

first = json.dumps(data, sort_keys=True)
second = json.dumps(data, sort_keys=True)
```

```
print 'UNSORTED MATCH:', unsorted == first
print 'SORTED MATCH :', first == second
```

排序后，人类查看结果会更为容易，而且还可以在测试中比较 JSON 输出。

```
$ python json_sort_keys.py
```

```
DATA: [{'a': 'A', 'c': 3.0, 'b': (2, 4)}]
JSON: [{"a": "A", "c": 3.0, "b": [2, 4]}]
SORT: [{"a": "A", "b": [2, 4], "c": 3.0}]
UNSORTED MATCH: False
SORTED MATCH : True
```

对于高度嵌套的数据结构，还可以指定一个缩进（indent）值，来得到格式美观的输出。

```
import json
```

```
data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0 } ]
print 'DATA:', repr(data)
```

```
print 'NORMAL:', json.dumps(data, sort_keys=True)
print 'INDENT:', json.dumps(data, sort_keys=True, indent=2)
```

当缩进是一个非负整数时，输出更类似于 pprint 的输出，数据结构中每一级的前导空格与其缩进级别匹配。

```
$ python json_indent.py
```

```
DATA: [{'a': 'A', 'c': 3.0, 'b': (2, 4)}]
NORMAL: [{"a": "A", "b": [2, 4], "c": 3.0}]
INDENT: [
    {
        "a": "A",
        "b": [
            2,
            4
        ],
        "c": 3.0
    }
]
```

不过，这样的详细输出会增加传输等量数据所需的字节数，所以生产环境中往往不使用这种输出。实际上，可以调整编码输出中分隔数据的设置，使之比默认格式更为紧凑。

```
import json
```

```
data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0 } ]
print 'DATA:', repr(data)
```

```
print 'repr(data)          :', len(repr(data))
```

```

plain_dump = json.dumps(data)
print 'dumps(data)          :', len(plain_dump)

small_indent = json.dumps(data, indent=2)
print 'dumps(data, indent=2) :', len(small_indent)

with_separators = json.dumps(data, separators=(',', ':'))
print 'dumps(data, separators):', len(with_separators)

```

`dumps()` 的分隔符 (`separators`) 参数应当是一个元组, 其中包含用来分隔列表中各项的字符串, 以及分隔字典中键和值的字符串。默认为 `(',', ':')`。通过去除空白符, 可以生成一个更为紧凑的输出。

```

$ python json_compact_encoding.py

DATA: [{'a': 'A', 'c': 3.0, 'b': (2, 4)}]
repr(data)          : 35
dumps(data)         : 35
dumps(data, indent=2) : 76
dumps(data, separators): 29

```

12.9.3 编码字典

JSON 格式要求字典的键是字符串。如果一个字典以非串类型作为键, 对这个字典编码时, 会生成一个异常。(异常类型取决于是否加载了模块的纯 Python 版本还是快速的 C 版本, 不过异常无非是 `TypeError` 或 `ValueError`)。要想绕开这个限制, 一种办法是使用 `skipkeys` 参数告诉编码器跳过非串的键。

```

import json

data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0, ('d',): 'D tuple' } ]

print 'First attempt'
try:
    print json.dumps(data)
except (TypeError, ValueError), err:
    print 'ERROR:', err

print
print 'Second attempt'
print json.dumps(data, skipkeys=True)

```

此时不会产生一个异常, 而是会忽略非串键。

```

$ python json_skipkeys.py

First attempt
ERROR: keys must be a string

```




```
Second attempt
[{"a": "A", "c": 3.0, "b": [2, 4]}
```

12.9.4 处理定制类型

目前为止，所有例子都使用 Python 的内置类型，因为这些类型得到了 json 的内置支持。通常还需要对定制类编码，有两种办法可以做到。

给定以下需要编码的类：

```
class MyObj(object):
    def __init__(self, s):
        self.s = s
    def __repr__(self):
        return '<MyObj(%s)>' % self.s
```

对 MyObj 实例编码的简单方法是定义一个函数，将未知类型转换为已知类型。这个函数并不需要具体完成编码，它只是将一个对象转换为另一个对象。

```
import json
import json_myobj

obj = json_myobj.MyObj('instance value goes here')

print 'First attempt'
try:
    print json.dumps(obj)
except TypeError, err:
    print 'ERROR:', err

def convert_to_builtin_type(obj):
    print 'default(', repr(obj), ')'
    # Convert objects to a dictionary of their representation
    d = { '__class__':obj.__class__.__name__,
          '__module__':obj.__module__,
        }
    d.update(obj.__dict__)
    return d

print
print 'With default'
print json.dumps(obj, default=convert_to_builtin_type)
```

在 convert_to_builtin_type() 中，json 无法识别的类实例会转换为字典，其中包含足够的信息，如果程序可以访问所需的 Python 模块，则可以利用这些信息重新创建对象。

```
$ python json_dump_default.py
```

```
First attempt
```

```
ERROR: <MyObj(instance value goes here)> is not JSON serializable
```

```
With default
default( <MyObj(instance value goes here)> )
{"s": "instance value goes here", "__module__": "json_myobj",
 "__class__": "MyObj"}
```

要对结果解码并创建一个 `MyObj()` 实例，可以使用 `loads()` 的 `object_hook` 参数关联解码器，从而可以从模块导入这个类，并用于创建实例。

对于从到来数据流解码的每个字典，都会调用 `object_hook`，这样就提供了一个机会，可以将字典转换为另外一种对象类型。`hook` 函数要返回调用应用所接收的对象而不是字典。

```
import json

def dict_to_object(d):
    if '__class__' in d:
        class_name = d.pop('__class__')
        module_name = d.pop('__module__')
        module = __import__(module_name)
        print 'MODULE:', module.__name__
        class_ = getattr(module, class_name)
        print 'CLASS:', class_
        args = dict( (key.encode('ascii'), value)
                     for key, value in d.items())
        print 'INSTANCE ARGS:', args
        inst = class_(**args)
    else:
        inst = d
    return inst

encoded_object = '''
[{"s": "instance value goes here",
  "__module__": "json_myobj", "__class__": "MyObj"}]
'''
myobj_instance = json.loads(encoded_object,
                             object_hook=dict_to_object)

print myobj_instance
```

由于 `json` 将串值转换为 `unicode` 对象，因此，用作类构造函数的 `keyword` 参数之前，需要将它们重新编码为 `ASCII` 串。

```
$ python json_load_object_hook.py
```

```
MODULE: json_myobj
CLASS: <class 'json_myobj.MyObj'>
INSTANCE ARGS: {'s': 'instance value goes here'}
[<MyObj(instance value goes here)>]
```

对于内置类型也有类似的 hook 函数，如整数 (parse_int)、浮点数 (parse_float) 和常量 (parse_constant)。

12.9.5 编码器和解码器类

除了之前介绍的便利函数外，json 模块还提供了一些类来完成编码和解码。直接使用这些类可以访问另外的 API 来定制其行为。

JSONEncoder 使用一个可迭代接口生成编码数据“块”，从而更容易写至文件或网络套接字，而不必在内存中表示完整的数据结构。

```
import json

encoder = json.JSONEncoder()
data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0 } ]

for part in encoder.iterencode(data):
    print 'PART:', part
```

输出按逻辑单元生成，而不是根据某个大小值。

```
$ python json_encoder_iterable.py
```

```
PART: [
PART: {
PART: "a"
PART: :
PART: "A"
PART: ,
PART: "c"
PART: :
PART: 3.0
PART: ,
PART: "b"
PART: :
PART: [2
PART: , 4
PART: ]
PART: }
PART: ]
```

encode() 方法基本上等价于表达式 ''.join(encoder.iterencode()) 生成的值，只不过之前会做一些额外的错误检查。

要对任意的对象编码，需要用一个类似于 convert_to_builtin_type() 中的实现覆盖 default() 方法。

```
import json
import json_myobj
```

```

class MyEncoder(json.JSONEncoder):

    def default(self, obj):
        print 'default(', repr(obj), ')'
        # Convert objects to a dictionary of their representation
        d = { '__class__':obj.__class__.__name__,
              '__module__':obj.__module__,
              }
        d.update(obj.__dict__)
        return d

obj = json_myobj.MyObj('internal data')
print obj
print MyEncoder().encode(obj)

```

输出与前一个实现的输出相同。

```

$ python json_encoder_default.py
<MyObj(internal data)>
default( <MyObj(internal data)> )
{"s": "internal data", "__module__": "json_myobj", "__class__":
"MyObj"}

```

这里要解码文本，然后将字典转换为一个对象，与前面的实现相比，这需要多做一些工作，不过不算太多。

```

import json

class MyDecoder(json.JSONDecoder):

    def __init__(self):
        json.JSONDecoder.__init__(self,
                                   object_hook=self.dict_to_object)

    def dict_to_object(self, d):
        if '__class__' in d:
            class_name = d.pop('__class__')
            module_name = d.pop('__module__')
            module = __import__(module_name)
            print 'MODULE:', module.__name__
            class_ = getattr(module, class_name)
            print 'CLASS:', class_
            args = dict( (key.encode('ascii'), value)
                        for key, value in d.items())
            print 'INSTANCE ARGS:', args
            inst = class_(**args)
        else:
            inst = d
        return inst

```



```
encoded_object = '''
[{"s": "instance value goes here",
  "__module__": "json_myobj", "__class__": "MyObj"}]
'''
```

```
myobj_instance = MyDecoder().decode(encoded_object)
print myobj_instance
```

输出与前面的例子相同。

```
$ python json_decoder_object_hook.py
```

```
MODULE: json_myobj
CLASS: <class 'json_myobj.MyObj'>
INSTANCE ARGS: {'s': 'instance value goes here'}
[<MyObj(instance value goes here)>]
```

12.9.6 处理流和文件

到目前为止，所有例子都假设整个数据结构的编码版本可以一次完全放在内存中。对于很大的数据结构，更合适的做法可能是将编码直接写至一个类文件的对象。便利函数 `load()` 和 `dump()` 会接受一个类文件对象的引用，用于读写。

```
import json
from StringIO import StringIO

data = [ { 'a': 'A', 'b': (2, 4), 'c': 3.0 } ]

f = StringIO()
json.dump(data, f)

print f.getvalue()
```

类似于这个例子中使用的 `StringIO` 缓冲区，也可以使用套接字或常规的文件句柄。

```
$ python json_dump_file.py
```

```
[{"a": "A", "c": 3.0, "b": [2, 4]}]
```

尽管没有优化，即一次只读取数据的一部分，但 `load()` 函数还提供了一个好处，它封装了从流输入生成对象的逻辑。

```
import json
from StringIO import StringIO

f = StringIO('{"a": "A", "c": 3.0, "b": [2, 4]}')
print json.load(f)
```

类似于 `dump()`，任何类文件对象都可以传递到 `load()`。

```
$ python json_load_file.py
```

```
[{'a': 'A', 'c': 3.0, 'b': [2, 4]}]
```

12.9.7 混合数据流

JSONDecoder 包含一个 `raw_decode()` 方法，如果一个数据结构后面跟有更多数据，如带尾部文本的 JSON 数据，可以用这个方法完成解码。返回值是对输入数据解码创建的对象，以及该数据的一个索引（指示解码从哪里结束）。

```
import json

decoder = json.JSONDecoder()
def get_decoded_and_remainder(input_data):
    obj, end = decoder.raw_decode(input_data)
    remaining = input_data[end:]
    return (obj, end, remaining)

encoded_object = '[{"a": "A", "c": 3.0, "b": [2, 4]}]'
extra_text = 'This text is not JSON.'

print 'JSON first:'
data = ' '.join([encoded_object, extra_text])
obj, end, remaining = get_decoded_and_remainder(data)

print 'Object          :', obj
print 'End of parsed input :', end
print 'Remaining text    :', repr(remaining)

print
print 'JSON embedded:'
try:
    data = ' '.join([extra_text, encoded_object, extra_text])
    obj, end, remaining = get_decoded_and_remainder(data)
except ValueError, err:
    print 'ERROR:', err
```

遗憾的是，这种做法只适用于对象出现在输入起始位置的情况。

```
$ python json_mixed_data.py
```

```
JSON first:
Object          : [{'a': 'A', 'c': 3.0, 'b': [2, 4]}]
End of parsed input : 35
Remaining text    : ' This text is not JSON.'

JSON embedded:
ERROR: No JSON object could be decoded
```

参见:

json (<http://docs.python.org/library/json.html>) 这个模块的标准库文档。

JavaScript Object Notation (<http://json.org/>) JSON 主页, 提供了文档以及其他语言的实现。

simplejson (<http://code.google.com/p/simplejson/>) simplejson, 由 Bob Ippolito 等人编写, 这是 Python 2.6 及以后版本包含的 json 库外部维护开发版本。它保持了与 Python 2.4 和 Python 2.5 的向后兼容性。

jsonpickle (<http://code.google.com/p/jsonpickle/>) jsonpickle 允许将任意 Python 对象串行化为 JSON。

12.10 xmlrpclib——XML-RPC 的客户端库

作用: XML-RPC 通信的客户端库。

Python 版本: 2.2 及以后版本

XML-RPC 是一个轻量级远程过程调用协议, 建立在 HTTP 和 XML 之上。xmlrpclib 模块允许 Python 程序与使用任何语言编写的 XML-RPC 服务器通信。

本节中的所有例子都使用了 xmlrpclib_server.py 中定义的服务器, 可以在源发布包中找到, 这里给出这个服务器以供参考。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
from xmlrpclib import Binary
import datetime

server = SimpleXMLRPCServer(('localhost', 9000),
                             logRequests=True,
                             allow_none=True)
server.register_introspection_functions()
server.register_multicall_functions()

class ExampleService:

    def ping(self):
        """Simple function to respond when called
        to demonstrate connectivity.
        """
        return True

    def now(self):
        """Returns the server current date and time."""
        return datetime.datetime.now()

    def show_type(self, arg):
        """Illustrates how types are passed in and out of
        server methods.
```



```

    Accepts one argument of any type.
    Returns a tuple with string representation of the value,
    the name of the type, and the value itself.
    """
    return (str(arg), str(type(arg)), arg)

def raises_exception(self, msg):
    "Always raises a RuntimeError with the message passed in"
    raise RuntimeError(msg)

def send_back_binary(self, bin):
    """Accepts single Binary argument, and unpacks and
    repacks it to return it."""
    data = bin.data
    response = Binary(data)
    return response

server.register_instance(ExampleService())

try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'

```

12.10.1 连接服务器

要将一个客户连接到服务器，最简单的方法是实例化一个 `ServerProxy` 对象，为它指定服务的 URI。例如，演示服务器在 `localhost` 的端口 9000 上运行。

```

import xmlrpclib

server = xmlrpclib.ServerProxy('http://localhost:9000')
print 'Ping:', server.ping()

```

在这种情况下，服务的 `ping()` 方法没有任何参数，它会返回一个布尔值。

```
$ python xmlrpclib_ServerProxy.py
```

```
Ping: True
```

还可以有其他选项来支持其他类型的传输。HTTP 和 HTTPS 已经明确得到支持，二者都提供基本验证。要实现一个新的通信通道，只需要一个新的传输类。例如，可以在 SMTP 之上实现 XML-RPC，这可能是一个很有意思的练习。

```

import xmlrpclib

server = xmlrpclib.ServerProxy('http://localhost:9000', verbose=True)
print 'Ping:', server.ping()

```


指定 `verbose` 选项会提供调试信息，这对于解决通信错误会很有用。

```
$ python xmlrpclib_ServerProxy_verbose.py
```

```
Ping: connect: (localhost, 9000)
connect fail: ('localhost', 9000)
connect: (localhost, 9000)
connect fail: ('localhost', 9000)
connect: (localhost, 9000)
send: 'POST /RPC2 HTTP/1.0\r\nHost: localhost:9000\r\nUser-Agent:
xmlrpclib.py/1.0.1 (by www.pythonware.com)\r\nContent-Type: text
/xml\r\nContent-Length: 98\r\n\r\n'
send: "<?xml version='1.0'?>\n<methodCall>\n<methodName>ping</met
hodName>\n<params>\n</params>\n</methodCall>\n"
reply: 'HTTP/1.0 200 OK\r\n'
header: Server: BaseHTTP/0.3 Python/2.5.1
header: Date: Sun, 06 Jul 2008 19:56:13 GMT
header: Content-type: text/xml
header: Content-length: 129
body: "<?xml version='1.0'?>\n<methodResponse>\n<params>\n<param
>\n<value><boolean>1</boolean></value>\n</param>\n</params>\n</m
ethodResponse>\n"
True
```

如果需要其他系统，可以将默认编码由 UTF-8 改为其他编码。

```
import xmlrpclib
```

```
server = xmlrpclib.ServerProxy('http://localhost:9000',
                               encoding='ISO-8859-1')
print 'Ping:', server.ping()
```

服务器会自动检测正确的编码。

```
$ python xmlrpclib_ServerProxy_encoding.py
```

```
Ping: True
```

`allow_none` 选项会控制 Python 的 `None` 值是自动转换为 `nil` 值还是会导致一个错误。

```
import xmlrpclib
```

```
server = xmlrpclib.ServerProxy('http://localhost:9000',
                               allow_none=True)
print 'Allowed:', server.show_type(None)
```

```
server = xmlrpclib.ServerProxy('http://localhost:9000',
                               allow_none=False)
```

```
try:
    server.show_type(None)
except TypeError as err:
```

```
print 'ERROR:', err
```

如果客户不允许 None，会在本地产生一个错误，不过如果未配置为允许 None，也可能从服务器产生错误。

```
$ python xmlrpclib_ServerProxy_allow_none.py
```

```
Allowed: ['None', "<type 'NoneType'>", None]
```

```
ERROR: cannot marshal None unless allow_none is enabled
```

12.10.2 数据类型

XML-RPC 协议能够识别一组有限的常用数据类型。这些类型可以作为参数或返回值传递，还可以结合来创建更为复杂的数据结构。

```
import xmlrpclib
import datetime
```

```
server = xmlrpclib.ServerProxy('http://localhost:9000')
```

```
for t, v in [ ('boolean', True),
              ('integer', 1),
              ('float', 2.5),
              ('string', 'some text'),
              ('datetime', datetime.datetime.now()),
              ('array', ['a', 'list']),
              ('array', ('a', 'tuple')),
              ('structure', {'a': 'dictionary'})
            ]:
    as_string, type_name, value = server.show_type(v)
    print '%-12s:' % t, as_string
    print '%12s' % '', type_name
    print '%12s' % '', value
```

简单的类型包括：

```
$ python xmlrpclib_types.py
```

```
boolean      : True
               <type 'bool'>
               True
integer      : 1
               <type 'int'>
               1
float        : 2.5
               <type 'float'>
               2.5
string       : some text
               <type 'str'>
```



```

        some text
datetime    : 20101128T20:15:21
              <type 'instance'>
              20101128T20:15:21
array       : ['a', 'list']
              <type 'list'>
              ['a', 'list']
array       : ['a', 'tuple']
              <type 'list'>
              ['a', 'tuple']
structure   : {'a': 'dictionary'}
              <type 'dict'>
              {'a': 'dictionary'}

```

所支持的这些类型可以嵌套，以创建任意复杂度的值。

```

import xmlrpclib
import datetime
import pprint

server = xmlrpclib.ServerProxy('http://localhost:9000')

data = { 'boolean': True,
         'integer': 1,
         'floating-point number': 2.5,
         'string': 'some text',
         'datetime': datetime.datetime.now(),
         'array': ['a', 'list'],
         'array': ('a', 'tuple'),
         'structure': {'a': 'dictionary'},
       }

arg = []
for i in range(3):
    d = {}
    d.update(data)
    d['integer'] = i
    arg.append(d)

print 'Before:'
pprint.pprint(arg)

print
print 'After:'
pprint.pprint(server.show_type(arg)[-1])

```

这个程序向示例服务器传递一个字典列表，其中包含所有支持的类型，由示例服务器返回数据。元组会转换为列表，datetime 实例转换为 DateTime 对象。否则，数据不变。

```
$ python xmlrpclib_types_nested.py
```

Before:

```
[{'array': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2008, 7, 6, 16, 24, 52, 348849),
  'floating-point number': 2.5,
  'integer': 0,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2008, 7, 6, 16, 24, 52, 348849),
  'floating-point number': 2.5,
  'integer': 1,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2008, 7, 6, 16, 24, 52, 348849),
  'floating-point number': 2.5,
  'integer': 2,
  'string': 'some text',
  'structure': {'a': 'dictionary'}}]
```

After:

```
[{'array': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20080706T16:24:52' at a5be18>,
  'floating-point number': 2.5,
  'integer': 0,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20080706T16:24:52' at a5bf30>,
  'floating-point number': 2.5,
  'integer': 1,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20080706T16:24:52' at a5bf80>,
  'floating-point number': 2.5,
  'integer': 2,
  'string': 'some text',
  'structure': {'a': 'dictionary'}}]
```

XML-RPC 支持日期作为一个内置类型，xmlrpclib 可以使用两个类在发出代理中表示日期

值，或者从服务器接收时表示日期值。默认地，会使用 DateTime 的一个内部版本，不过如果设置了 use_datetime 选项，则会打开 datetime 支持，使用 datetime 模块中的类。

12.10.3 传递对象

Python 类实例被处理为结构，并作为字典传递，对象的属性将作为字典中的值。

```
import xmlrpclib
import pprint

class MyObj:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __repr__(self):
        return 'MyObj(%s, %s)' % (repr(self.a), repr(self.b))

server = xmlrpclib.ServerProxy('http://localhost:9000')

o = MyObj(1, 'b goes here')
print 'o :', o
pprint.pprint(server.show_type(o))
o2 = MyObj(2, o)
print 'o2 :', o2
pprint.pprint(server.show_type(o2))
```

值从服务器发送回客户时，结果将是客户上的一个字典，因为值中没有编码相关信息来告诉服务器（或客户）应当将它实例化为类的一部分。

```
$ python xmlrpclib_types_object.py

o : MyObj(1, 'b goes here')
  [{"a": 1, "b": "b goes here"}, "<type 'dict'>",
   {"a": 1, "b": "b goes here"}]
o2 : MyObj(2, MyObj(1, 'b goes here'))
  [{"a": 2, "b": {"a": 1, "b": "b goes here"}},
   "<type 'dict'>",
   {"a": 2, "b": {"a": 1, "b": "b goes here"}}]
```

12.10.4 二进制数据

所有传递到服务器的值都会编码，并自动转义。不过，有些数据类型包含的字符可能不是合法的 XML。例如，二进制图像数据可能包括 ASCII 控制范围 0~31 中的字节值。要传递二进制数据，最好使用 Binary 类对其编码来进行传输。

```
import xmlrpclib

server = xmlrpclib.ServerProxy('http://localhost:9000')
```

```

s = 'This is a string with control characters' + '\0'
print 'Local string:', s

data = xmlrpclib.Binary(s)
print 'As binary:', server.send_back_binary(data)

try:
    print 'As string:', server.show_type(s)
except xmlrpclib.Fault as err:
    print '\nERROR:', err

```

如果将一个包含 NULL 字节的串传递到 show_type(), XML 解析器会产生一个异常。

```
$ python xmlrpclib_Binary.py
```

```

Local string: This is a string with control characters
As binary: This is a string with control characters
As string:
ERROR: <Fault 1: "<class 'xml.parsers.expat.ExpatError':not
well-formed (invalid token): line 6, column 55">

```

还可以利用 pickle 使用 Binary 对象发送对象。通过网络向可执行代码发送大量数据时, 通常存在一些安全问题, 这里也不例外 (也就是说, 除非通信通道是安全的, 否则不要这么做)。

```

import xmlrpclib
import cPickle as pickle
import pprint

class MyObj:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __repr__(self):
        return 'MyObj(%s, %s)' % (repr(self.a), repr(self.b))

server = xmlrpclib.ServerProxy('http://localhost:9000')

o = MyObj(1, 'b goes here')
print 'Local:', id(o)
print o

print '\nAs object:'
pprint.pprint(server.show_type(o))

p = pickle.dumps(o)
b = xmlrpclib.Binary(p)
r = server.send_back_binary(b)

```

```
o2 = pickle.loads(r.data)
print '\nFrom pickle:', id(o2)
pprint.pprint(o2)
```

Binary 实例的数据属性包含对象的 pickle 版本，所以在使用之前必须解除 pickle。这会得到一个不同的对象（有一个新的 id 值）。

```
$ python xmlrpclib_Binary_pickle.py

Local: 4321077872
MyObj(1, 'b goes here')

As object:
["{'a': 1, 'b': 'b goes here'}", "<type 'dict'>",
 {'a': 1, 'b': 'b goes here'}]

From pickle: 4321252344
MyObj(1, 'b goes here')
```

12.10.5 异常处理

由于 XML-RPC 服务器可以用任何语言编写，所以不能直接传输异常类。实际上，服务器中产生的异常会转换为 Fault 对象，并在客户端作为异常在本地产生。

```
import xmlrpclib

server = xmlrpclib.ServerProxy('http://localhost:9000')
try:
    server.raises_exception('A message')
except Exception, err:
    print 'Fault code:', err.faultCode
    print 'Message   :', err.faultString
```

原来的错误消息保存在 faultString 属性中，fault-Code 设置为一个 XML-RPC 错误码。

```
$ python xmlrpclib_exception.py

Fault code: 1
Message   : <type 'exceptions.RuntimeError':A message
```

12.10.6 将调用结合在一个消息中

多调用 (Multicall) 是对 XML-RPC 协议的扩展，它允许同时发送多个调用，并收集响应，返回给调用者。Python 2.4 在 xmlrpclib 中增加了 MultiCall 类。

```
import xmlrpclib

server = xmlrpclib.ServerProxy('http://localhost:9000')
```

```

multicall = xmlrpclib.MultiCall(server)
multicall.ping()
multicall.show_type(1)
multicall.show_type('string')

```

```

for i, r in enumerate(multicall()):
    print i, r

```

要使用 MultiCall 实例，可以像 ServerProxy 一样调用 MultiCall 的方法，然后不带参数地调用这个对象来具体运行远程函数。返回值是一个迭代器，可以得到所有调用的结果。

```
$ python xmlrpclib_MultiCall.py
```

```

0 True
1 ['1', "<type 'int'>", 1]
2 ['string', "<type 'str'>", 'string']

```

如果某个调用导致一个 Fault，从迭代器生成结果时会产生异常，相应地不再生成更多结果。

```
import xmlrpclib
```

```
server = xmlrpclib.ServerProxy('http://localhost:9000')
```

```

multicall = xmlrpclib.MultiCall(server)
multicall.ping()
multicall.show_type(1)
multicall.raises_exception('Next to last call stops execution')
multicall.show_type('string')

```

```

try:
    for i, r in enumerate(multicall()):
        print i, r
except xmlrpclib.Fault as err:
    print 'ERROR:', err

```

由于第 3 个响应（来自 raises_exception()）生成了一个异常，所以无法再访问 show_type() 的响应。

```
$ python xmlrpclib_MultiCall_exception.py
```

```

0 True
1 ['1', "<type 'int'>", 1]
ERROR: <Fault 1: "<type 'exceptions.RuntimeError'>:Next to last call
stops execution">

```

参见：

xmlrpclib (<http://docs.python.org/lib/module-xmlrpclib.html>) 这个模块的标准库文档。
SimpleXMLRPCServer (12.11 节) 一个 XML-RPC 服务器实现。

12.11 SimpleXMLRPCServer——一个 XML-RPC 服务器

作用：实现一个 XML-RPC 服务器。

Python 版本：2.2 及以后版本

SimpleXMLRPCServer 模块包含有一些类，可以使用 XML-RPC 协议创建跨平台、语言独立的服务器。除了 Python 之外，还有很多其他语言的客户端库，这使得 XML-RPC 成为构建 RPC 式服务的一个便利选择。

注意：这里提供的所有例子还包含一个客户端模块与演示服务器交互。要运行这些例子，需要两个单独的 shell 窗口，一个运行服务器，另一个运行客户端。

12.11.1 一个简单的服务器

这个简单的服务器示例提供了一个函数，它取一个字典名，并返回这个字典的内容。第一步是创建 SimpleXMLRPCServer 实例，告诉它在哪里监听到来的请求（这里要在 'localhost' 的端口 9000 监听）。下一步定义一个函数作为服务的一部分，并注册这个函数，使服务器知道如何调用该函数。最后一步是将这个服务器放在一个接收和响应请求的无限循环中。

警告：这个实现存在明显的安全隐患。如果服务器位于开放的因特网或安全问题可能导致严重后果的环境中，不要在这样的服务器上运行这个实现。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import logging
import os

# Set up logging
logging.basicConfig(level=logging.DEBUG)

server = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)

# Expose a function
def list_contents(dir_name):
    logging.debug('list_contents(%s)', dir_name)
    return os.listdir(dir_name)
server.register_function(list_contents)

# Start the server
try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'
```

使用 xmlrpclib 的客户类，可以在 URL `http://localhost:9000` 访问这个服务器。这个示例代码展示了如何从 Python 调用 `list_contents()` 服务。

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
print proxy.list_contents('/tmp')
```

`ServerProxy` 使用基 URL 连接到服务器，然后在代理上直接调用方法。代理上调用的各个方法会转换为对服务器的一个请求。参数使用 XML 格式化，然后通过一个 POST 消息发送到服务器。服务器解包 XML，根据从客户调用的方法名来确定调用哪个函数。参数将传递到这个函数，返回值将转换回 XML，以便返回给客户。

启动服务器会得到：

```
$ python SimpleXMLRPCServer_function.py
```

Use Control-C to exit

在第二个窗口运行客户，会显示 `/tmp` 目录的内容。

```
$ python SimpleXMLRPCServer_function_client.py
```

```
['.s.PGSQL.5432', '.s.PGSQL.5432.lock', '.X0-lock', '.X11-unix',
'ccc_exclude.1mkahl', 'ccc_exclude.BKG3gb', 'ccc_exclude.M5jrgo',
'ccc_exclude.SPecwL', 'com.hp.launchport', 'emacs527',
'hsperfdata_dhellmann', 'launch-8hGHUp', 'launch-RQnlcc',
'launch-trsdly', 'launchd-242.T5UzTy', 'var_backups']
```

完成这个请求之后，日志输出会出现在服务器窗口中。

```
$ python SimpleXMLRPCServer_function.py
```

Use Control-C to exit

```
DEBUG:root:list_contents(/tmp)
```

```
localhost -- [29/Jun/2008 09:32:07] "POST /RPC2 HTTP/1.0" 200 -
```

输出的第 1 行来自 `list_contents()` 中的 `logging.debug()` 调用。第 2 行来自记录请求的服务器，因为 `logRequests` 为 `True`。

12.11.2 备用 API 名

有时，模块或库中使用的函数名并不是外部 API 中要使用的名。函数名之所以有变化，可能是因为加载了一个平台特定的实现，或者服务 API 要根据一个配置文件动态地构建，也可能实际函数要用桩函数替换来完成测试。要注册一个有备用名的函数，需要将这个名作为第 2 个参数传递到 `register_function()`，如下所示。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import os
```

```

server = SimpleXMLRPCServer(('localhost', 9000))

# Expose a function with an alternate name
def list_contents(dir_name):
    return os.listdir(dir_name)
server.register_function(list_contents, 'dir')

try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'

客户现在应当使用 dir() 而不是 list_contents()。

import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
print 'dir():', proxy.dir('/tmp')
try:
    print '\nlist_contents():', proxy.list_contents('/tmp')
except xmlrpclib.Fault as err:
    print '\nERROR:', err

```

调用 list_contents() 会得到一个错误，因为服务器上不再有以这个名字注册的处理程序。

```
$ python SimpleXMLRPCServer_alternate_name_client.py
```

```

dir(): ['ccc_exclude.GIqLcR', 'ccc_exclude.kzR42t',
'ccc_exclude.LV04nf', 'ccc_exclude.Vfzylm', 'emacs527',
'icssuis527', 'launch-9hTTwf', 'launch-kCXjtT',
'launch-Nwc3AB', 'launch-pwCgej', 'launch-Xrku4Q',
'launch-YtDZBJ', 'launchd-167.AfaNuZ', 'var_backups']

list_contents():
ERROR: <Fault 1: '<type \'exceptions.Exception\'>:method
"list_contents" is not supported'>

```

12.11.3 加点的 API 名

还可以用通常情况下不能作为合法 Python 标识符的名字来注册单个函数。例如，可以在名字中包含一个点号 (.) 分隔服务中的命名空间。下面的例子扩展了“目录”服务，增加了“创建”和“删除”调用。所有函数注册时都使用了前缀“dir.”，这样一来，同一个服务器可以使用不同的前缀提供其他服务。这个例子中还有一点不同，有些函数会返回 None，所以必须告诉服务器将 None 值转换为一个 nil 值。

```

from SimpleXMLRPCServer import SimpleXMLRPCServer
import os

```

```

server = SimpleXMLRPCServer(('localhost', 9000), allow_none=True)

server.register_function(os.listdir, 'dir.list')
server.register_function(os.mkdir, 'dir.create')
server.register_function(os.rmdir, 'dir.remove')

try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'

```

要在客户中调用服务函数，只需用加点的名称指示函数。

```

import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
print 'BEFORE      :', 'EXAMPLE' in proxy.dir.list('/tmp')
print 'CREATE      :', proxy.dir.create('/tmp/EXAMPLE')
print 'SHOULD EXIST :', 'EXAMPLE' in proxy.dir.list('/tmp')
print 'REMOVE      :', proxy.dir.remove('/tmp/EXAMPLE')
print 'AFTER       :', 'EXAMPLE' in proxy.dir.list('/tmp')

```

假设当前系统上没有 /tmp/EXAMPLE 文件，示例客户脚本的输出如下。

```

$ python SimpleXMLRPCServer_dotted_name_client.py
BEFORE      : False
CREATE      : None
SHOULD EXIST : True
REMOVE      : None
AFTER       : False

```

12.11.4 任意 API 名

还有一个有趣的特性，能够用一些非法的 Python 对象属性名来注册函数。下面的示例服务用名字“multiply args”注册了一个函数。

```

from SimpleXMLRPCServer import SimpleXMLRPCServer

server = SimpleXMLRPCServer(('localhost', 9000))

def my_function(a, b):
    return a * b
server.register_function(my_function, 'multiply args')

try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'

```

由于所注册的名字包含一个空格，因此不能使用点记法直接从代理访问。不过，可以使用 `getattr()`。

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
print getattr(proxy, 'multiply args')(5, 5)
```

但是要避免用类似这样的名字创建服务。提供这个例子并不是因为这种想法很好，而是因为确实存在这种有任意名的服务，可能需要新程序才能够调用这些服务。

```
$ python SimpleXMLRPCServer_arbitrary_name_client.py
```

25

12.11.5 公布对象的方法

前面几节讨论了使用好的命名约定和命名空间机制建立 API 的技术。要在 API 中结合命名空间，另一种方法是使用类实例并公布其方法。可以使用仅有一个方法的实例重新创建第一个例子。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)

class DirectoryService:
    def list(self, dir_name):
        return os.listdir(dir_name)

server.register_instance(DirectoryService())

try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'
```

客户可以直接调用这个方法，如下所示。

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
print proxy.list('/tmp')
```

输出如下：

```
$ python SimpleXMLRPCServer_instance_client.py
```



```
['ccc_exclude.lmkahl', 'ccc_exclude.BKG3gb', 'ccc_exclude.M5jrgo',
'ccc_exclude.SPecwL', 'com.hp.launchport', 'emacs527',
'hsperfdata_dhellmann', 'launch-8hGHUp', 'launch-RQnlcc',
'launch-trsdly', 'launchd-242.T5UzTy', 'var_backups']
```

不过，服务的“dir.”前缀没有丢失。可以定义一个类来建立一个服务树（可以从客户调用）加以恢复。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)

class ServiceRoot:
    pass

class DirectoryService:
    def list(self, dir_name):
        return os.listdir(dir_name)

root = ServiceRoot()
root.dir = DirectoryService()

server.register_instance(root, allow_dotted_names=True)

try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'
```

通过注册 ServiceRoot 实例，并启用 allow_dotted_names，当一个请求到来时，服务器有权限遍历这个对象树从而使用 getattr() 查找指定的方法。

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
print proxy.dir.list('/tmp')

dir.list() 的输出与之前实现的输出相同。

$ python SimpleXMLRPCServer_instance_dotted_names_client.py

['ccc_exclude.lmkahl', 'ccc_exclude.BKG3gb', 'ccc_exclude.M5jrgo',
'ccc_exclude.SPecwL', 'com.hp.launchport', 'emacs527',
'hsperfdata_dhellmann', 'launch-8hGHUp', 'launch-RQnlcc',
'launch-trsdly', 'launchd-242.T5UzTy', 'var_backups']
```

12.11.6 分派调用

默认情况下, `register_instance()` 会查找实例的所有可调用属性 (属性名以一个下划线 (“_”) 开头的除外), 并用其名字注册。对于公布的方法, 为了更谨慎, 可以使用定制的分派逻辑, 如下例所示。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)

def expose(f):
    "Decorator to set exposed flag on a function."
    f.exposed = True
    return f

def is_exposed(f):
    "Test whether another function should be publicly exposed."
    return getattr(f, 'exposed', False)

class MyService:
    PREFIX = 'prefix'

    def _dispatch(self, method, params):
        # Remove our prefix from the method name
        if not method.startswith(self.PREFIX + '.'):
            raise Exception('method "%s" is not supported' % method)

        method_name = method.partition('.')[2]
        func = getattr(self, method_name)
        if not is_exposed(func):
            raise Exception('method "%s" is not supported' % method)

        return func(*params)

    @expose
    def public(self):
        return 'This is public'

    def private(self):
        return 'This is private'

server.register_instance(MyService())
try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
```

```
print 'Exiting'
```

MyService 的 public() 方法标志为要公布到 XML-RPC 服务，而 private() 不公布。客户试图访问 MyService 中的一个函数时会调用 _dispatch() 方法。它首先强制使用一个前缀（在这里是“prefix.”，不过也可以使用任意的串）。然后要求这个函数有一个名为 exposed 的属性，而且值为 true。为方便起见，这里使用一个修饰符在函数上设置这个 exposed 标志。

以下是一些示例客户调用。

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
print 'public():', proxy.prefix.public()

try:
    print 'private():', proxy.prefix.private()
except Exception, err:
    print '\nERROR:', err

try:
    print 'public() without prefix:', proxy.public()
except Exception, err:
    print '\nERROR:', err
```

下面给出得到的输出，这里捕获并报告了预料中的错误消息。

```
$ python SimpleXMLRPCServer_instance_with_prefix_client.py
```

```
public(): This is public
private():
ERROR: <Fault 1: ' <type \'exceptions.Exception\'>:method
"prefix.private" is not supported'>
public() without prefix:
ERROR: <Fault 1: ' <type \'exceptions.Exception\'>:method
"public" is not supported'>
```

还有另外一些方法来覆盖分派机制，包括直接从 SimpleXMLRPCServer 派生子类。可参考这个模块中的 docstring 来了解更多详细内容。

12.11.7 自省 API

与很多网络服务一样，可以查询一个 XML-RPC 服务器来询问它支持哪些方法，并了解如何使用这些方法。SimpleXMLRPCServer 包括一组公共方法来完成这个自省。默认情况下，这些方法是关闭的，不过可以用 `register_introspection_functions()` 启用。通过在服务类上定义 `_listMethods()` 和 `_methodHelp()`，可以在服务中增加对 `system.listMethods()` 和 `system.methodHelp()` 的支持。

```
from SimpleXMLRPCServer import ( SimpleXMLRPCServer,
                                list_public_methods,
                                .
import os
```



```

import inspect

server = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)
server.register_introspection_functions()

class DirectoryService:

    def _listMethods(self):
        return list_public_methods(self)

    def _methodHelp(self, method):
        f = getattr(self, method)
        return inspect.getdoc(f)

    def list(self, dir_name):
        """list(dir_name) => [<filenames>]

        Returns a list containing the contents of
        the named directory.
        """
        return os.listdir(dir_name)

server.register_instance(DirectoryService())

try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'

```

在这里，便利函数 `list_public_methods()` 扫描一个实例，返回不是以下划线（`_`）开头的可调用属性名。重新定义 `_listMethods()` 来应用所需的规则。类似地，对于这个基本例子，`_methodHelp()` 返回了函数的 docstring，不过也可以写为从其他来源构建一个帮助文本串。

这个客户会查询服务器，并报告所有可公开调用的方法。

```

import xmlrpclib

proxy = xmlrpclib.ServerProxy('http://localhost:9000')
for method_name in proxy.system.listMethods():
    print '=' * 60
    print method_name
    print '-' * 60
    print proxy.system.methodHelp(method_name)
    print

```

结果中还包含了系统方法。

```
$ python SimpleXMLRPCServer_introspection_client.py
```

```
=====
list
-----
```

```
list(dir_name) => [<filenames>]
```

Returns a list containing the contents of the named directory.

```
=====
system.listMethods
-----
```

```
system.listMethods() => ['add', 'subtract', 'multiple']
```

Returns a list of the methods supported by the server.

```
=====
system.methodHelp
-----
```

```
system.methodHelp('add') => "Adds two integers together"
```

Returns a string containing documentation for the specified method.

```
=====
system.methodSignature
-----
```

```
system.methodSignature('add') => [double, int, int]
```

Returns a list describing the signature of the method. In the above example, the add method takes two integers as arguments and returns a double result.

This server does NOT support system.methodSignature.

参见:

SimpleXMLRPCServer(<http://docs.python.org/lib/module-SimpleXMLRPCServer.html>) 这个模块的标准库文档。

XML-RPC How To(<http://www.tldp.org/HOWTO/XML-RPC-HOWTO/index.html>) 描述如何使用 XML-RPC 采用多种语言实现客户和服务端。

XML-RPC Extensions (<http://ontosys.com/xml-rpc/extensions.php>) 指定了 XML-RPC 协议的一个扩展。

xmrlpplib (12.10 节) XML-RPC 客户库。

第 13 章

Email

Email 是数字通信最古老的形式之一，不过也是最流行的形式之一。Python 的标准库提供了发送、接收和存储 Email 消息的模块。

smtpplib 与邮件服务器通信来传送消息。smtpd 可以用于创建定制的邮件服务器，它提供了一些很有用的类，可以在其他应用中调试 Email 传输。

imaplib 使用 IMAP 协议管理存储在服务器上的消息。它为 IMAP 客户提供了一个底层 API，可以查询、获取、移动和删除消息。

利用 mailbox，可以使用多种标准格式创建和修改本地消息归档，包括流行的 mbox 和 Maildir 格式，这是很多 Email 客户程序使用的格式。

13.1 smtpplib——简单邮件传输协议客户

作用：与 SMTP 服务器交互，包括发送 Email。

Python 版本: 1.5.2 及以后版本

`smtplib` 包括一个 `SMTP` 类，可以用来与邮件服务器通信发送邮件。

注意：在后面的例子中，Email 地址、主机名和 IP 地址都故意修改为没有实际意义。除此以外，这些脚本准确地展示了命令和响应序列。

13.1.1 发送 Email 消息

SMTP 最常见的用法就是连接到邮件服务器发送消息。可以把邮件服务器主机名和端口传递到构造函数，也可以显式调用 `connect()`。一旦连接，可以调用 `sendmail()` 并提供信封参数和消息体。消息文本要完整并遵循 RFC 2882，因为 `smtplib` 根本不会修改内容或首部。这说明，调用者需要添加 `From` 和 `To` 首部。

[illegible]

```

msg['From'] = email.utils.formataddr(('Author',
                                     'author@example.com'))
msg['Subject'] = 'Simple test message'

server = smtplib.SMTP('mail')
server.set_debuglevel(True) # show communication with the server
try:
    server.sendmail('author@example.com',
                   ['recipient@example.com'],
                   msg.as_string())
finally:
    server.quit()

```

这个例子还打开了调试，以显示客户与服务器之间的通信。否则，这个示例根本不会产生任何输出。

```
$ python smtplib_sendmail.py
```

```

send: 'ehlo farnsworth.local\r\n'
reply: '250-mail.example.com Hello [192.168.1.27], pleased to meet y
ou\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-SIZE\r\n'
reply: '250-DSN\r\n'
reply: '250-ETRN\r\n'
reply: '250-AUTH GSSAPI DIGEST-MD5 CRAM-MD5\r\n'
reply: '250-DELIVERBY\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: mail.example.com Hello [192.168.1.27], pl
eased to meet you
ENHANCEDSTATUSCODES
PIPELINING
8BITMIME
SIZE
DSN
ETRN
AUTH GSSAPI DIGEST-MD5 CRAM-MD5
DELIVERBY
HELP
send: 'mail FROM:<author@example.com> size=229\r\n'
reply: '250 2.1.0 <author@example.com>... Sender ok\r\n'
reply: retcode (250); Msg: 2.1.0 <author@example.com>... Sender ok
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: '250 2.1.5 <recipient@example.com>... Recipient ok\r\n'
reply: retcode (250); Msg: 2.1.5 <recipient@example.com>... Recipien
t ok

```

```

send: 'data\r\n'
reply: '354 Enter mail, end with "." on a line by itself\r\n'
reply: retcode (354); Msg: Enter mail, end with "." on a line by its
elf
data: (354, 'Enter mail, end with "." on a line by itself')
send: 'Content-Type: text/plain; charset="us-ascii"\r\nMIME-Version:
1.0\r\nContent-Transfer-Encoding: 7bit\r\nTo: Recipient <recipient@
example.com>\r\nFrom: Author <author@example.com>\r\nSubject: Simple
test message\r\n\r\nThis is the body of the message.\r\n.\r\n'
reply: '250 2.0.0 oAT1TiRA010200 Message accepted for delivery\r\n'
reply: retcode (250); Msg: 2.0.0 oAT1TiRA010200 Message accepted for
delivery
data: (250, '2.0.0 oAT1TiRA010200 Message accepted for delivery')
send: 'quit\r\n'
reply: '221 2.0.0 mail.example.com closing connection\r\n'
reply: retcode (221); Msg: 2.0.0 mail.example.com closing connection

```

sendmail() 的第 2 个参数, 即接收者, 会作为一个列表传递。这个列表中 can 包括任意多个地址, 将把消息按顺序逐个传送到各个地址。由于信封信息与消息首部是分开的, 所以通过把地址包含在方法参数中而不是置于消息首部, 可以实现暗送 (blind carbon copy, BCC)。

13.1.2 认证和加密

SMTP 类还会处理认证和传输层安全 (transport layer security, TLS) 加密 (如果服务器提供支持)。要确定服务器是否支持 TLS, 可以直接调用 ehlo() 为服务器标识客户, 询问可以得到哪些扩展。然后调用 has_extn() 来检查结果。启动 TLS 之后, 在认证之前必须再次调用 ehlo()。

```

import smtplib
import email.utils
from email.mime.text import MIMEText
import getpass

# Prompt the user for connection info
to_email = raw_input('Recipient: ')
servername = raw_input('Mail server name: ')
username = raw_input('Mail username: ')
password = getpass.getpass("%s's password: " % username)

# Create the message
msg = MIMEText('Test message from PyMOTW.')
msg.set_unixfrom('author')
msg['To'] = email.utils.formataddr(('Recipient', to_email))
msg['From'] = email.utils.formataddr(('Author',
                                     'author@example.com'))
msg['Subject'] = 'Test from PyMOTW'

```

```

server = smtplib.SMTP(servername)
try:
    server.set_debuglevel(True)

    # identify ourselves, prompting server for supported features
    server.ehlo()

    # If we can encrypt this session, do it
    if server.has_extn('STARTTLS'):
        server.starttls()
        server.ehlo() # reidentify ourselves over TLS connection

    server.login(username, password)
    server.sendmail('author@example.com',
                    [to_email],
                    msg.as_string())
finally:
    server.quit()

```

启用 TLS 之后, STARTTLS 扩展不会出现在对 EHLO 的应答中。

```
$ python smtplib_authenticated.py
```

```

Recipient: recipient@example.com
Mail server name: smtpauth.isp.net
Mail username: user@isp.net
user@isp.net's password:
send: 'ehlo localhost.local\r\n'
reply: '250-elasmtip-isp.net Hello localhost.local [<your IP here>]\r\n'
reply: '250-SIZE 14680064\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-AUTH PLAIN LOGIN CRAM-MD5\r\n'
reply: '250-STARTTLS\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: elasmtip-isp.net Hello localhost.local [<y
our IP here>]
SIZE 14680064
PIPELINING
AUTH PLAIN LOGIN CRAM-MD5
STARTTLS
HELP
send: 'STARTTLS\r\n'
reply: '220 TLS go ahead\r\n'
reply: retcode (220); Msg: TLS go ahead
send: 'ehlo localhost.local\r\n'
reply: '250-elasmtip-isp.net Hello localhost.local [<your IP here>]\r\n'

```

```
reply: '250-SIZE 14680064\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-AUTH PLAIN LOGIN CRAM-MD5\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: elasmtp-isp.net Hello farnsworth.local [<
your
IP here>]
SIZE 14680064
PIPELINING
AUTH PLAIN LOGIN CRAM-MD5
HELP
send: 'AUTH CRAM-MD5\r\n'
reply: '334 PDExNjkyLjEyMjI1Nz1AZWxhc210cCltZWfseS5hdGwuc2EuZWfy
dGhsa
W5rLm5ldD4=\r\n'
reply: retcode (334); Msg: PDExNjkyLjEyMjI1Nz1AZWxhc210cCltZWfse
S5hdG
wuc2EuZWfydGhsaW5rLm5ldD4=
send: 'ZGhlbGxtYW5uQGVhcnRobGlualy5uZXQgN2Q1YjAyYTRmMGQ1YzZjM2NjOTNjZ
DclMD
QxN2ViYjg=\r\n'
reply: '235 Authentication succeeded\r\n'
reply: retcode (235); Msg: Authentication succeeded
send: 'mail FROM:<author@example.com> size=221\r\n'
reply: '250 OK\r\n'
reply: retcode (250); Msg: OK
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: '250 Accepted\r\n'
reply: retcode (250); Msg: Accepted
send: 'data\r\n'
reply: '354 Enter message, ending with "." on a line by itself\r\n'
reply: retcode (354); Msg: Enter message, ending with "." on a line
by itself
data: (354, 'Enter message, ending with "." on a line by itself')
send: 'Content-Type: text/plain; charset="us-ascii"\r\nMIME-Version:
1.0\r\nContent-Transfer-Encoding: 7bit\r\nTo: Recipient
<recipient@example.com>\r\nFrom: Author <author@example.com>\r\nSubj
ect: Test
from PyMOTW\r\n\r\nTest message from PyMOTW.\r\n.\r\n'
reply: '250 OK id=1KjxNj-00032a-Ux\r\n'
reply: retcode (250); Msg: OK id=1KjxNj-00032a-Ux
data: (250, 'OK id=1KjxNj-00032a-Ux')
send: 'quit\r\n'
reply: '221 elasmtp-isp.net closing connection\r\n'
reply: retcode (221); Msg: elasmtp-isp.net closing connection
```

13.1.3 验证 Email 地址

SMTP 协议包括一个命令来询问服务器地址是否合法。通常，VRFY 是禁用的，以避免垃圾邮件工具（spammer）查找到合法的 Email 地址。不过，如果启用这个命令，客户可以询问服务器一个地址是否合法，并接收一个状态码指示其合法性，同时提供用户的全名（如果有）。

```
import smtplib

server = smtplib.SMTP('mail')
server.set_debuglevel(True) # show communication with the server
try:
    dhellmann_result = server.verify('dhellmann')
    notthere_result = server.verify('notthere')
finally:
    server.quit()

print 'dhellmann:', dhellmann_result
print 'notthere :', notthere_result
```

如输出中最后两行所示，地址 dhellmann 是合法的，而 notthere 不合法。

```
$ python smtplib_verify.py
```

```
send: 'vrfy <dhellmann>\r\n'
reply: '250 2.1.5 Doug Hellmann <dhellmann@mail.example.com>\r\n'
reply: retcode (250); Msg: 2.1.5 Doug Hellmann <dhellmann@mail.example.com>
send: 'vrfy <notthere>\r\n'
reply: '550 5.1.1 <notthere>... User unknown\r\n'
reply: retcode (550); Msg: 5.1.1 <notthere>... User unknown
send: 'quit\r\n'
reply: '221 2.0.0 mail.example.com closing connection\r\n'
reply: retcode (221); Msg: 2.0.0 mail.example.com closing connection
dhellmann: (250, '2.1.5 Doug Hellmann <dhellmann@mail.example.com>')
notthere : (550, '5.1.1 <notthere>... User unknown')
```

参见：

smtplib (<http://docs.python.org/lib/module-smtplib.html>) 这个模块的标准库文档。

RFC 821 (<http://tools.ietf.org/html/rfc821.html>) 简单邮件传输协议（Simple Mail Transfer Protocol, SMTP）规范。

RFC 1869 (<http://tools.ietf.org/html/rfc1869.html>) 基本协议的 SMTP 服务扩展。

RFC 822 (<http://tools.ietf.org/html/rfc822.html>) “ARPA Internet 文本消息格式标准”，原来的 Email 消息格式规范。

RFC 2822 (<http://tools.ietf.org/html/rfc2822.html>) “Internet 消息格式”更新为 Email 消息格式。

email 解析 Email 消息的标准库模块。

smtpd (13.2 节) 实现一个简单的 SMTP 服务器。

13.2 smtpd——示例邮件服务器

作用：包含一些实现 SMTP 服务器的类。

Python 版本：2.1 及以后版本

smtpd 模块包括一些类来构建简单的邮件传输协议服务器。这是 smtplib 使用的协议的服务器端。

13.2.1 邮件服务器基类

已经提供的所有示例服务器的基类都是 SMTPServer。它会处理与客户的通信以及接收到的数据，并提供了一个方便的 hook，可以覆盖这个 hook，从而一旦得到完整的消息就可以处理消息。

构造函数参数包括本地地址和远程地址：要在本地地址监听连接，代理消息将传送到这个远程地址。方法 process_message() 提供为一个 hook，要由派生类覆盖。接收到完整的消息时会调用这个方法，并指定以下参数：

peer

客户的地址，这是一个包含 IP 和到来端口的元组。

mailfrom

消息信封以外的“from”信息，传送消息时由客户提供给服务器。这个消息不一定总与 From 首部匹配。

rcpttos

消息信封中的接收者列表。同样地，这个列表不一定总与 To 首部匹配，特别是秘密送给接收者时。

data

完整的 RFC 2822 消息体。

process_message() 的默认实现只产生一个 NotImplementedError 异常。下面的例子定义了一个子类，它覆盖了这个方法，将打印接收到的消息的有关信息。

```
import smtpd
import asyncore

class CustomSMTPServer(smtpd.SMTPServer):

    def process_message(self, peer, mailfrom, rcpttos, data):
        print 'Receiving message from:', peer
        print 'Message addressed from:', mailfrom
        print 'Message addressed to   :', rcpttos
        print 'Message length          :', len(data)
        return
```

```
server = CustomSMTPServer(('127.0.0.1', 1025), None)
```

```
asyncore.loop()
```

SMTPServer 使用了 `asyncore`，所以要运行服务器，需要调用 `asyncore.loop()`。

还需要一个客户来展示服务器。可以修改 `smtplib` 一节中的某个例子来创建一个客户，向在端口 1025 本地运行的测试服务器发送数据。

```
import smtplib
import email.utils
from email.mime.text import MIMEText

# Create the message
msg = MIMEText('This is the body of the message.')
msg['To'] = email.utils.formataddr(('Recipient',
                                   'recipient@example.com'))
msg['From'] = email.utils.formataddr(('Author',
                                     'author@example.com'))
msg['Subject'] = 'Simple test message'

server = smtplib.SMTP('127.0.0.1', 1025)
server.set_debuglevel(True) # show communication with the server
try:
    server.sendmail('author@example.com',
                   ['recipient@example.com'],
                   msg.as_string())
finally:
    server.quit()
```

要测试这些程序，可以在一个终端窗口运行 `smtpd_custom.py`，在另一个终端窗口运行 `smtpd_senddata.py`。

```
$ python smtpd_custom.py
```

```
Receiving message from: ('127.0.0.1', 58541)
Message addressed from: author@example.com
Message addressed to   : ['recipient@example.com']
Message length         : 229
```

`smtpd_senddata.py` 的调试输出显示了与服务器的所有通信。

```
$ python smtpd_senddata.py
```

```
send: 'ehlo farnsworth.local\r\n'
reply: '502 Error: command "EHLO" not implemented\r\n'
reply: retcode (502); Msg: Error: command "EHLO" not implemented
send: 'helo farnsworth.local\r\n'
reply: '250 farnsworth.local\r\n'
reply: retcode (250); Msg: farnsworth.local
```

```

send: 'mail FROM:<author@example.com>\r\n'
reply: '250 Ok\r\n'
reply: retcode (250); Msg: Ok
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: '250 Ok\r\n'
reply: retcode (250); Msg: Ok
send: 'data\r\n'
reply: '354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: End data with <CR><LF>.<CR><LF>
data: (354, 'End data with <CR><LF>.<CR><LF>')
send: 'Content-Type: text/plain; charset="us-ascii"\r\nMIME-Version:
1.0\r\n
Content-Transfer-Encoding: 7bit\r\nTo: Recipient <recipient@example.
com>\r\n
From: Author <author@example.com>\r\nSubject: Simple test message\r\
n\r\nThis
is the body of the message.\r\n.\r\n'
reply: '250 Ok\r\n'
reply: retcode (250); Msg: Ok
data: (250, 'Ok')
send: 'quit\r\n'
reply: '221 Bye\r\n'
reply: retcode (221); Msg: Bye

```

要停止服务器，只需按下 Ctrl-C。

13.2.2 调试服务器

前面的例子显示了 `process_message()` 的参数，不过 `smtpd` 还包括一个专门设计的服务器，名为 `DebuggingServer`，用来完成更完备的调试。它会把到来的消息完整地打印到控制台，然后停止处理（它不会把消息转发给一个真正的邮件服务器）。

```

import smtpd
import asyncore

```

```

server = smtpd.DebuggingServer(('127.0.0.1', 1025), None)

```

```

asyncore.loop()

```

使用前面的 `smtpd_senddata.py` 客户程序，以下是 `DebuggingServer` 的输出。

```

$ python smtpd_debug.py

```

```

----- MESSAGE FOLLOWS -----
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: Recipient <recipient@example.com>

```

```

From: Author <author@example.com>
Subject: Simple test message
X-Peer: 127.0.0.1

This is the body of the message.
----- END MESSAGE -----

```

13.2.3 代理服务器

PureProxy 类实现了一个简单的代理服务器。到来的消息将作为构造函数的一个参数，向上转发给服务器。

警告：smtpd 的标准库文档指出，“运行这个模块时，很有可能进入一种开放转发[⊖]（open relay）状态，所以请务必谨慎”。

建立代理服务器的步骤与建立调试服务器的步骤类似。

```

import smtpd
import asyncore

server = smtpd.PureProxy(('127.0.0.1', 1025), ('mail', 25))

asyncore.loop()

```

不过，它不会打印任何输出，所以要验证是否正常工作，需要查看邮件服务器日志。

```

Oct 19 19:16:34 homer sendmail[6785]: m9JNGXJb006785:
from=<author@example.com>, size=248, class=0, nrcpts=1,
msgid=<200810192316.m9JNGXJb006785@homer.example.com>,
proto=ESMTP, daemon=MTA, relay=[192.168.1.17]

```

参见：

smtpd (<http://docs.python.org/lib/module-smtpd.html>) 这个模块的标准库文档。

smtplib (13.1 节) 提供一个客户接口。

email 解析 Email 消息。

asyncore (11.4 节) 编写异步服务器的基本模块。

RFC 2822 (<http://tools.ietf.org/html/rfc2822.html>) 定义 Email 消息格式。

13.3 imaplib——IMAP4 客户库

作用：完成 IMAP4 通信的客户库。

Python 版本：1.5.2 及以后版本

imaplib 实现了一个可与 IMAP 4 服务器通信的客户，IMAP 表示 Internet 消息访问协议

[⊖] 也称为匿名转发。——译者注

(Internet Message Access Protocol)。IMAP 协议定义了一组发送到服务器的命令，以及发回客户的响应。大多数命令都可以作为 IMAP4 对象的方法得到（IMAP4 对象用于与服务器通信）。

下面的例子将讨论 IMAP 协议的一部分，不过并不完备。要想全面地了解有关的详细信息，可以参考 RFC 3501。

13.3.1 变种

有 3 个客户类可以使用不同的机制与服务器通信。第 1 个是 IMAP4，它使用明文套接字；第 2 个是 IMAP4_SSL，使用基于 SSL 套接字的加密通信；最后 1 个是 IMAP4_stream，使用一个外部命令的标准输入和标准输出。这里的所有例子都使用 IMAP4_SSL，不过其他类的 API 也是类似的。

13.3.2 连接到服务器

要建立一个 IMAP 服务器的连接，有 2 个步骤。首先，建立套接字连接本身。其次，用服务器上的一个账户作为用户完成认证。下面的示例代码会从一个配置文件读取服务器和用户信息。

```
import imaplib
import ConfigParser
import os

def open_connection(verbose=False):
    # Read the config file
    config = ConfigParser.ConfigParser()
    config.read([os.path.expanduser('~/.pymotw')])

    # Connect to the server
    hostname = config.get('server', 'hostname')
    if verbose: print 'Connecting to', hostname
    connection = imaplib.IMAP4_SSL(hostname)

    # Login to our account
    username = config.get('account', 'username')
    password = config.get('account', 'password')
    if verbose: print 'Logging in as', username
    connection.login(username, password)
    return connection

if __name__ == '__main__':
    c = open_connection(verbose=True)
    try:
        print c
    finally:
        c.logout()
```



运行时, `open_connection()` 从用户主目录中的一个文件读取配置信息, 然后打开 IMAP4_SSL 连接并认证。

```
$ python imaplib_connect.py
```

```
Connecting to mail.example.com
Logging in as example
<imaplib.IMAP4_SSL instance at 0x928cb0>
```

本节的其他例子还会重用这个模块, 以避免重复代码。

认证失败

如果建立了连接, 但是认证失败, 会产生一个异常。

```
import imaplib
import ConfigParser
import os

# Read the config file
config = ConfigParser.ConfigParser()
config.read([os.path.expanduser('~/.pymotw')])

# Connect to the server
hostname = config.get('server', 'hostname')
print 'Connecting to', hostname
connection = imaplib.IMAP4_SSL(hostname)

# Login to our account
username = config.get('account', 'username')
password = 'this_is_the_wrong_password'
print 'Logging in as', username
try:
    connection.login(username, password)
except Exception as err:
    print 'ERROR:', err
```

这个例子故意用错误的密码来触发这个异常。

```
$ python imaplib_connect_fail.py
```

```
Connecting to mail.example.com
Logging in as example
ERROR: Authentication failed.
```

13.3.3 示例配置

示例账户有 3 个邮箱: INBOX、Archive 和 2008 (Archive 的一个子文件夹)。邮箱的层次结构如下:

- INBOX

- Archive

- 2008

INBOX 文件夹下有一个未读的消息，Archive/2008 中有一个已读的消息。

13.3.4 列出邮箱

要获取一个账户的可用邮箱，可以使用 list() 方法。

```
import imaplib
from pprint import pprint
from imaplib_connect import open_connection

c = open_connection()
try:
    typ, data = c.list()
    print 'Response code:', typ
    print 'Response:'
    pprint(data)
finally:
    c.logout()
```

返回值是一个 tuple，其中包含一个响应码，以及由服务器返回的数据。除非出现一个错误，否则响应码都是 OK。list() 的数据是一个字符串序列，其中包含标志、层次结构定界符和每个邮箱的邮箱名。

```
$ python imaplib_list.py

Response code: OK
Response:
['(\\HasNoChildren) "." INBOX',
 '(\\HasChildren) "." "Archive"',
 '(\\HasNoChildren) "." "Archive.2008"']
```

可以使用 re 或 csv 将各个响应串划分为 3 个部分（参见本节最后参考资料中的 IMAP Backup Script，其中给出了一个使用 csv 的例子）。

```
import imaplib
import re

from imaplib_connect import open_connection

list_response_pattern = re.compile(
    r'\\((?P<flags>.*?)\\) "(?P<delimiter>.*)" (?P<name>.*)'
)

def parse_list_response(line):
    match = list_response_pattern.match(line)
    flags, delimiter, mailbox_name = match.groups()
```

```

mailbox_name = mailbox_name.strip('\"')
return (flags, delimiter, mailbox_name)

if __name__ == '__main__':
    c = open_connection()
    try:
        typ, data = c.list()
    finally:
        c.logout()
    print 'Response code:', typ

    for line in data:
        print 'Server response:', line
        flags, delimiter, mailbox_name = parse_list_response(line)
        print 'Parsed response:', (flags, delimiter, mailbox_name)

```

如果邮箱名包含空格，服务器会对邮箱名加引号，不过以后在对服务器的其他调用中使用邮箱名时需要将这些引号去除。

```
$ python imaplib_list_parse.py
```

```

Response code: OK
Server response: (\HasNoChildren) "." INBOX
Parsed response: ('\\HasNoChildren', '.', 'INBOX')
Server response: (\HasChildren) "." "Archive"
Parsed response: ('\\HasChildren', '.', 'Archive')
Server response: (\HasNoChildren) "." "Archive.2008"
Parsed response: ('\\HasNoChildren', '.', 'Archive.2008')

```

list() 有一些参数可以指定层次结构中的邮箱。例如，要列出 Archive 的子文件夹，需要传入 “Archive” 作为目录 (directory) 参数。

```

import imaplib

from imaplib_connect import open_connection

if __name__ == '__main__':
    c = open_connection()
    try:
        typ, data = c.list(directory='Archive')
    finally:
        c.logout()
    print 'Response code:', typ

    for line in data:
        print 'Server response:', line

```

这样只会返回一个子文件夹。


```
$ python imaplib_list_subfolders.py
```

```
Response code: OK
Server response: (\HasNoChildren) "." "Archive.2008"
```

或者，要列出与一个模式匹配的文件夹，需要传入模式（pattern）参数。

```
import imaplib

from imaplib_connect import open_connection
if __name__ == '__main__':
    c = open_connection()
    try:
        typ, data = c.list(pattern='*Archive*')
    finally:
        c.logout()
    print 'Response code:', typ

    for line in data:
        print 'Server response:', line
```

在这种情况下，Archive 和 Archive.2008 都会包含在响应中。

```
$ python imaplib_list_pattern.py
```

```
Response code: OK
Server response: (\HasChildren) "." "Archive"
Server response: (\HasNoChildren) "." "Archive.2008"
```

13.3.5 邮箱状态

使用 status() 可以询问内容的有关统计信息。表 13.1 列出了标准中定义的状态条件。

表 13.1 IMAP 4 邮箱状态条件

条 件	含 义
MESSAGES	邮箱中的消息数
RECENT	设置了 \Recent 标志的消息数
UIDNEXT	邮箱的下一个惟一标识符值
UIDVALIDITY	邮箱的惟一标识符合法性值
UNSEEN	未设置 \Seen 标志的消息数

状态条件必须格式化为用空格分隔的字符串，并包围在括号中，编码采用 IMAP4 规范中“列表”的编码。

```
import imaplib
import re
```

```

from imaplib_connect import open_connection
from imaplib_list_parse import parse_list_response
if __name__ == '__main__':
    c = open_connection()
    try:
        typ, data = c.list()
        for line in data:
            flags, delimiter, mailbox = parse_list_response(line)
            print c.status(
                mailbox,
                '(MESSAGES RECENT UIDNEXT UIDVALIDITY UNSEEN)')
    finally:
        c.logout()

```

返回值仍是 tuple，其中包含一个响应码和一个来自服务器的信息列表。在这里，列表中包含一个字符串，其格式为首先是邮箱名（用引号包围），然后是状态条件和值（用括号括起）。

```
$ python imaplib_status.py
```

```

('OK', ['"INBOX" (MESSAGES 1 RECENT 0 UIDNEXT 3 UIDVALIDITY
1222003700 UNSEEN 1)'])
('OK', ['"Archive" (MESSAGES 0 RECENT 0 UIDNEXT 1 UIDVALIDITY
1222003809 UNSEEN 0)'])
('OK', ['"Archive.2008" (MESSAGES 1 RECENT 0 UIDNEXT 2 UIDVALIDITY
1222003831 UNSEEN 0)'])

```

13.3.6 选择邮箱

一旦客户得到认证，基本操作模式为选择一个邮箱，然后向服务器询问邮箱中的消息。这个连接是有状态的，所以选择一个邮箱之后，所有命令都会处理该邮箱中的消息，直至选择一个新邮箱。

```

import imaplib
import imaplib_connect

c = imaplib_connect.open_connection()
try:
    typ, data = c.select('INBOX')
    print typ, data
    num_msgs = int(data[0])
    print 'There are %d messages in INBOX' % num_msgs
finally:
    c.close()
    c.logout()

```

响应数据包含邮箱中的消息总数。

```
$ python imaplib_select.py
```



```
OK ['1']
There are 1 messages in INBOX
```

如果指定了一个不合法的邮箱，响应码为 NO。

```
import imaplib
import imaplib_connect

c = imaplib_connect.open_connection()
try:
    typ, data = c.select('Does Not Exist')
    print typ, data
finally:
    c.logout()
```

数据中包含一个描述问题的错误消息。

```
$ python imaplib_select_invalid.py

NO ["Mailbox doesn't exist: Does Not Exist"]
```

13.3.7 搜索消息

选择邮箱之后，可以使用 `search()` 来获取邮箱中消息的 ID。

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response

c = imaplib_connect.open_connection()
try:
    typ, mailbox_data = c.list()
    for line in mailbox_data:
        flags, delimiter, mailbox_name = parse_list_response(line)
        c.select(mailbox_name, readonly=True)
        typ, msg_ids = c.search(None, 'ALL')
        print mailbox_name, typ, msg_ids
finally:
    try:
        c.close()
    except:
        pass
    c.logout()
```

消息 ID 由服务器分配，并依赖于具体实现。IMAP4 协议实现了两种 ID，一种是事务期间给定时刻消息的顺序 ID，另一种是消息的 UID 标识符，并对二者做了严格区分，不过并非所有服务器都同时实现了这两类 ID。

```
$ python imaplib_search_all.py

INBOX OK ['1']
```

```
Archive OK ['']
Archive.2008 OK ['1']
```

在这里, INBOX 和 Archive.2008 分别有一个 id 为 1 的不同消息。其他邮箱为空。

13.3.8 搜索规则

还可以使用很多其他搜索规则, 包括查看消息的日期、标志和其他首部。要全面了解有关详细信息, 可以参考 RFC 3501 的 6.4.4 节。

要查找主题中包含 “test message 2” 的消息, 应当如下构造搜索规则:

```
(SUBJECT "test message 2")
```

这个例子会查找所有邮箱中标题为 “test message 2” 的消息。

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response

c = imaplib_connect.open_connection()
try:
    typ, mailbox_data = c.list()
    for line in mailbox_data:
        flags, delimiter, mailbox_name = parse_list_response(line)
        c.select(mailbox_name, readonly=True)
        typ, msg_ids = c.search(None, '(SUBJECT "test message 2")')
        print mailbox_name, typ, msg_ids
finally:
    try:
        c.close()
    except:
        pass
    c.logout()
```

这个账户中只有一个这样的消息, 位于 INBOX 中。

```
$ python imaplib_search_subject.py
```

```
INBOX OK ['1']
Archive OK ['']
Archive.2008 OK ['']
```

搜索规则还可以结合。

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response

c = imaplib_connect.open_connection()
try:
```



```

typ, mailbox_data = c.list()
for line in mailbox_data:
    flags, delimiter, mailbox_name = parse_list_response(line)
    c.select(mailbox_name, readonly=True)
    typ, msg_ids = c.search(
        None,
        '(FROM "Doug" SUBJECT "test message 2")')
    print mailbox_name, typ, msg_ids
finally:
    try:
        c.close()
    except:
        pass
    c.logout()

```

这里用一个逻辑与（and）操作来结合搜索规则。

```
$ python imaplib_search_from.py
```

```

INBOX OK ['1']
Archive OK ['']
Archive.2008 OK ['']

```

13.3.9 获取消息

使用 `fetch()` 方法，可以利用 `search()` 返回的标识符来获取消息的内容（或部分内容），以便做进一步处理。这个方法有两个参数：要获取的消息和所获取的消息部分。

`message_ids` 参数是一个用逗号分隔的 id 列表（例如，“1”，“1,2”）或者是一个 id 区间（如 1:2）。`message_parts` 参数是一个消息段名 IMAP 列表。与 `search()` 的搜索规则类似，IMAP 协议指定了命名消息段，所以客户可以高效地获取他们真正需要的那部分消息。例如，要获取一个邮箱中消息的首部，可以使用 `fetch()` 并指定参数 `BODY.PEEK[HEADER]`。

注意：还可以使用另一种方法获取首部（`BODY[HEADERS]`），不过这种形式有一个副作用，会隐式地将消息标志为已读，而在很多情况下我们并不希望如此。

```

import imaplib
import pprint
import imaplib_connect

imaplib.Debug = 4
c = imaplib_connect.open_connection()
try:
    c.select('INBOX', readonly=True)
    typ, msg_data = c.fetch('1', '(BODY.PEEK[HEADER] FLAGS)')
    pprint.pprint(msg_data)
finally:

```

```

try:
    c.close()
except:
    pass
c.logout()

```

fetch() 的返回值已经部分解析, 所以与 list() 的返回值相比, 从某种程度上讲会更难处理。可以打开调试, 显示客户与服务器之间完整的交互, 来理解为什么会这样。

```
$ python imaplib_fetch_raw.py
```

```

13:12.54 imaplib version 2.58
13:12.54 new IMAP4 connection, tag=CFKH
13:12.54 < * OK dovecot ready.
13:12.54 > CFKH0 CAPABILITY
13:12.54 < * CAPABILITY IMAP4rev1 SORT THREAD=REFERENCES MULTIAPPEND
    UNSELECT IDLE CHILDREN LISTEXT LIST-SUBSCRIBED NAMESPACE AUTH=PLAIN
13:12.54 < CFKH0 OK Capability completed.
13:12.54 CAPABILITIES: ('IMAP4REV1', 'SORT', 'THREAD=REFERENCES', 'M
    ULTIAPPEND', 'UNSELECT', 'IDLE', 'CHILDREN', 'LISTEXT', 'LIST-SUBSCR
    IBED', 'NAMESPACE', 'AUTH=PLAIN')
13:12.54 > CFKH1 LOGIN example "password"
13:13.18 < CFKH1 OK Logged in.
13:13.18 > CFKH2 EXAMINE INBOX
13:13.20 < * FLAGS (\Answered \Flagged \Deleted \Seen \Draft $NotJun
    k $Junk)
13:13.20 < * OK [PERMANENTFLAGS {}] Read-only mailbox.
13:13.20 < * 2 EXISTS
13:13.20 < * 1 RECENT
13:13.20 < * OK [UNSEEN 1] First unseen.
13:13.20 < * OK [UIDVALIDITY 1222003700] UIDs valid
13:13.20 < * OK [UIDNEXT 4] Predicted next UID
13:13.20 < CFKH2 OK [READ-ONLY] Select completed.
13:13.20 > CFKH3 FETCH 1 (BODY.PEEK[HEADER] FLAGS)
13:13.20 < * 1 FETCH (FLAGS ($NotJunk) BODY[HEADER] {595}
13:13.20 read literal size 595
13:13.20 < )
13:13.20 < CFKH3 OK Fetch completed.
13:13.20 > CFKH4 CLOSE
13:13.21 < CFKH4 OK Close completed.
13:13.21 > CFKH5 LOGOUT
13:13.21 < * BYE Logging out
13:13.21 BYE response: Logging out
13:13.21 < CFKH5 OK Logout completed.
'1 (FLAGS ($NotJunk) BODY[HEADER] {595}',
'Return-Path: <dhellmann@example.com>\r\nReceived: from example.com
(localhost [127.0.0.1])\r\n\tby example.com (8.13.4/8.13.4) with ESM
TP id m8LDTGW4018260\r\n\tfor <example@example.com>; Sun, 21 Sep 200

```

```

8 09:29:16 -0400\r\nReceived: (from dhellmann@localhost)\r\n\tby example.com (8.13.4/8.13.4/Submit) id m8LDTGZ5018259\r\n\tfor example@example.com; Sun, 21 Sep 2008 09:29:16 -0400\r\nDate: Sun, 21 Sep 2008 09:29:16 -0400\r\nFrom: Doug Hellmann <dhellmann@example.com>\r\nMessage-Id: <200809211329.m8LDTGZ5018259@example.com>\r\nTo: example@example.com\r\nSubject: test message 2\r\n\r\n'),
)']

```

FETCH 命令的响应中，首先是标志，然后指示有 595 字节的首部数据。客户用这个消息响应构造一个元组，然后用一个包含右括号 (“)”) 的字符串（服务器在获取命令响应的最后会发送这个字符串）结束这个序列。由于采用了这种格式，就能更容易地单独获取信息的不同部分，或者重新组合响应并在客户端解析。

```

import imaplib
import pprint
import imaplib_connect

c = imaplib_connect.open_connection()
try:
    c.select('INBOX', readonly=True)

    print 'HEADER:'
    typ, msg_data = c.fetch('1', '(BODY.PEEK[HEADER])')
    for response_part in msg_data:
        if isinstance(response_part, tuple):
            print response_part[1]

    print 'BODY TEXT:'
    typ, msg_data = c.fetch('1', '(BODY.PEEK[TEXT])')
    for response_part in msg_data:
        if isinstance(response_part, tuple):
            print response_part[1]

    print '\nFLAGS:'
    typ, msg_data = c.fetch('1', '(FLAGS)')
    for response_part in msg_data:
        print response_part
        print imaplib.ParseFlags(response_part)
finally:
    try:
        c.close()
    except:
        pass
    c.logout()

```

单独地获取值还有一个额外的好处，这样更易于使用 ParseFlags() 解析响应中的标志。

```
$ python imaplib_fetch_separately.py
```

```

HEADER:
Return-Path: <dhellmann@example.com>
Received: from example.com (localhost [127.0.0.1])
    by example.com (8.13.4/8.13.4) with ESMTP id m8LDTGW4018260
    for <example@example.com>; Sun, 21 Sep 2008 09:29:16 -0400
Received: (from dhellmann@localhost)
    by example.com (8.13.4/8.13.4/Submit) id m8LDTGZ5018259
    for example@example.com; Sun, 21 Sep 2008 09:29:16 -0400
Date: Sun, 21 Sep 2008 09:29:16 -0400
From: Doug Hellmann <dhellmann@example.com>
Message-Id: <200809211329.m8LDTGZ5018259@example.com>
To: example@example.com
Subject: test message 2

```

```

BODY TEXT:
second message

```

```

FLAGS:
1 (FLAGS ($NotJunk))
('$NotJunk',)

```

13.3.10 完整消息

如前所述，客户可以向服务器单独请求消息中的单个部分。还可以获取整个消息（采用 RFC 2822 规范格式化的邮件消息），并用 `email` 模块的类进行解析。

```

import imaplib
import email
import imaplib_connect

c = imaplib_connect.open_connection()
try:
    c.select('INBOX', readonly=True)

    typ, msg_data = c.fetch('1', '(RFC822)')
    for response_part in msg_data:
        if isinstance(response_part, tuple):
            msg = email.message_from_string(response_part[1])
            for header in [ 'subject', 'to', 'from' ]:
                print '%-8s: %s' % (header.upper(), msg[header])

finally:
    try:
        c.close()
    except:

```



```
pass
c.logout()
```

利用 email 模块中的解析器，可以非常容易地访问和处理消息。下面的例子只打印了各个消息的一些首部。

```
$ python imaplib_fetch_rfc822.py

SUBJECT : test message 2
TO      : example@example.com
FROM    : Doug Hellmann <dhellmann@example.com>
```

13.3.11 上传消息

要向邮箱添加一个新消息，需要构造一个 Message 实例，并把它传递到 append() 方法，同时提供消息的时间戳。

```
import imaplib
import time
import email.message
import imaplib_connect

new_message = email.message.Message()
new_message.set_unixfrom('pymotw')
new_message['Subject'] = 'subject goes here'
new_message['From'] = 'pymotw@example.com'
new_message['To'] = 'example@example.com'
new_message.set_payload('This is the body of the message.\n')
print new_message

c = imaplib_connect.open_connection()
try:
    c.append('INBOX', '',
            imaplib.Time2Internaldate(time.time()),
            str(new_message))

    # Show the headers for all messages in the mailbox
    c.select('INBOX')
    typ, [msg_ids] = c.search(None, 'ALL')
    for num in msg_ids.split():
        typ, msg_data = c.fetch(num, '(BODY.PEEK[HEADER])')
        for response_part in msg_data:
            if isinstance(response_part, tuple):
                print '\n%s:' % num
                print response_part[1]

finally:
    try:
```

```
c.close()
except:
    pass
c.logout()
```

这个例子中使用的消息内容 (payload) 是一个简单的纯文本 Email 体。Message 还支持 MIME 编码的多部分消息。

```
pymotw
Subject: subject goes here
From: pymotw@example.com
To: example@example.com
```

This is the body of the message.

```
1:
Return-Path: <dhellmann@example.com>
Received: from example.com (localhost [127.0.0.1])
    by example.com (8.13.4/8.13.4) with ESMTP id m8LDTGW4018260
    for <example@example.com>; Sun, 21 Sep 2008 09:29:16 -0400
Received: (from dhellmann@localhost)
    by example.com (8.13.4/8.13.4/Submit) id m8LDTGZ5018259
    for example@example.com; Sun, 21 Sep 2008 09:29:16 -0400
Date: Sun, 21 Sep 2008 09:29:16 -0400
From: Doug Hellmann <dhellmann@example.com>
Message-Id: <200809211329.m8LDTGZ5018259@example.com>
To: example@example.com
Subject: test message 2
```

```
2:
Return-Path: <doug.hellmann@example.com>
Message-Id: <0D9C3C50-462A-4FD7-9E5A-11EE222D721D@example.com>
From: Doug Hellmann <doug.hellmann@example.com>
To: example@example.com
Content-Type: text/plain; charset=US-ASCII; format=flowed; delsp=yes
Content-Transfer-Encoding: 7bit
Mime-Version: 1.0 (Apple Message framework v929.2)
Subject: lorem ipsum
Date: Sun, 21 Sep 2008 12:53:16 -0400
X-Mailer: Apple Mail (2.929.2)
```

```
3:
pymotw
```

```
Subject: subject goes here
From: pymotw@example.com
To: example@example.com
```

13.3.12 移动和复制消息

一旦消息上传到服务器，则可以使用 `move()` 或 `copy()` 移动或复制，而无须下载。与 `fetch()` 一样，这些方法要处理消息 id 区间。

```
import imaplib
import imaplib_connect

c = imaplib_connect.open_connection()
try:
    # Find the "SEEN" messages in INBOX
    c.select('INBOX')
    typ, [response] = c.search(None, 'SEEN')
    if typ != 'OK':
        raise RuntimeError(response)
    # Create a new mailbox, "Archive.Today"
    msg_ids = ','.join(response.split(' '))
    typ, create_response = c.create('Archive.Today')
    print 'CREATED Archive.Today:', create_response

    # Copy the messages
    print 'COPYING:', msg_ids
    c.copy(msg_ids, 'Archive.Today')

    # Look at the results
    c.select('Archive.Today')
    typ, [response] = c.search(None, 'ALL')
    print 'COPIED:', response

finally:
    c.close()
    c.logout()
```

这个示例脚本在 `Archive` 下创建一个新的邮箱，并把已读消息从 `INBOX` 复制到这个邮箱。

```
$ python imaplib_archive_read.py
```

```
CREATED Archive.Today: ['Create completed.']
COPYING: 1,2
COPIED: 1 2
```

再次运行这个脚本，可以看出检查返回码的重要性。这里调用 `create()` 创建新邮箱时没有产生异常，而是会报告这个邮箱已经存在。

```
$ python imaplib_archive_read.py
```

```
CREATED Archive.Today: ['Mailbox exists.']
COPYING: 1,2
COPIED: 1 2 3 4
```

13.3.13 删除消息

尽管很多现代邮箱客户程序使用一个“垃圾文件夹”模型来处理已删除的消息，但是这些消息往往并没有移动到一个真正的文件夹中。实际上，删除邮件只会更新它们的标志，在其中添加 \Deleted。“清空”垃圾箱的操作是通过 EXPUNGE 命令实现的。下面这个示例脚本将查找主题包含“Lorem ipsum”的归档消息，设置已删除标志，然后再次查询服务器，可以看到这些消息仍在文件夹中。

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response

c = imaplib_connect.open_connection()
try:
    c.select('Archive.Today')

    # What ids are in the mailbox?
    typ, [msg_ids] = c.search(None, 'ALL')
    print 'Starting messages:', msg_ids

    # Find the message(s)
    typ, [msg_ids] = c.search(None, '(SUBJECT "Lorem ipsum"')
    msg_ids = ','.join(msg_ids.split(' '))
    print 'Matching messages:', msg_ids

    # What are the current flags?
    typ, response = c.fetch(msg_ids, '(FLAGS)')
    print 'Flags before:', response

    # Change the Deleted flag
    typ, response = c.store(msg_ids, '+FLAGS', r'(\Deleted)')

    # What are the flags now?
    typ, response = c.fetch(msg_ids, '(FLAGS)')
    print 'Flags after:', response

    # Really delete the message.
    typ, response = c.expunge()
    print 'Expunged:', response

    # What ids are left in the mailbox?
    typ, [msg_ids] = c.search(None, 'ALL')
```

```

print 'Remaining messages:', msg_ids

finally:
    try:
        c.close()
    except:
        pass
    c.logout()

```

显式地调用 `expunge()` 会删除消息，不过调用 `close()` 也能达到同样的效果。区别在于，调用 `close()` 时，客户不会得到删除通知。

```
$ python imaplib_delete_messages.py
```

```

Starting messages: 1 2 3 4
Matching messages: 1,3
Flags before: ['1 (FLAGS (\\Seen $NotJunk))', '3 (FLAGS (\\Seen
\\Recent $NotJunk))']
Flags after: ['1 (FLAGS (\\Deleted \\Seen $NotJunk))',
'3 (FLAGS (\\Deleted \\Seen \\Recent $NotJunk))']
Expunged: ['1', '2']
Remaining messages: 1 2

```

参见:

`imaplib` (<http://docs.python.org/library/imaplib.html>) 这个模块的标准库文档。

What is IMAP? (www.imap.org/about/whatisIMAP.html) `imap.org` 提供的 IMAP 协议描述。

University of Washington IMAP Information Center (<http://www.washington.edu/imap/>) 关于 IMAP 信息的一个很好的资源，还提供了源代码。

RFC 3501 (<http://tools.ietf.org/html/rfc3501.html>) Internet 消息访问协议 (Internet Message Access Protocol)。

RFC 2822 (<http://tools.ietf.org/html/rfc2822.html>) Internet 消息格式 (Internet Message Format)。

IMAP Backup Script (<http://snipplr.com/view/7955/imap-backup-script/>) 这个脚本可以用于备份 IMAP 服务器的邮件。

`rfc822` `rfc822` 模块包含一个 RFC 822/RFC 2822 解析器。

`email` `email` 模块用于解析邮件消息。

`mailbox` (13.4 节) 本地邮箱解析器。

`ConfigParser` (14.8 节) 读写配置文件。

`IMAPClient` (<http://imapclient.freshfoo.com/>) 这是一个与 IMAP 服务器通信的更高层客户程序，由 Menno Smits 编写。


```

                                'There are 3 lines.\n',
                                ]))

    mbox.add(msg)
    mbox.flush()

    msg = mailbox.mboxMessage()
    msg.set_unixfrom('author')
    msg['From'] = from_addr
    msg['To'] = to_addr
    msg['Subject'] = 'Sample message 2'
    msg.set_payload('This is the second body.\n')
    mbox.add(msg)
    mbox.flush()
finally:
    mbox.unlock()

print open('example.mbox', 'r').read()

```

这个脚本的结果是一个新的邮箱文件，其中包含两个邮件消息。

```
$ python mailbox_mbox_create.py
```

```

From MAILER-DAEMON Mon Nov 29 02:00:11 2010
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 1

```

```

This is the body.
>From (should be escaped).
There are 3 lines.

```

```

From MAILER-DAEMON Mon Nov 29 02:00:11 2010
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 2

```

```
This is the second body.
```

读取 mbox 邮箱

要读取一个已有的邮箱，需要打开这个邮箱，像字典一样处理这个 mbox 对象。键是邮箱实例定义的任意值，它们只作为消息对象的内部标识符，并不一定有实际意义。

```

import mailbox

mbox = mailbox.mbox('example.mbox')
for message in mbox:
    print message['subject']

```

打开的邮箱支持迭代器协议，不过与真正的字典对象不同，邮箱的默认迭代器会处理值（values）而不是键（keys）。

```
$ python mailbox_mbox_read.py
```

```
Sample message 1
Sample message 2
```

从 mbox 邮箱删除消息

要从一个 mbox 文件删除已有的消息，可以使用 `remove()` 并提供这个消息的键，也可以使用 `del`。

```
import mailbox

mbox = mailbox.mbox('example.mbox')
mbox.lock()
try:
    to_remove = []
    for key, msg in mbox.iteritems():
        if '2' in msg['subject']:
            print 'Removing:', key
            to_remove.append(key)
    for key in to_remove:
        mbox.remove(key)
finally:
    mbox.flush()
    mbox.close()

print open('example.mbox', 'r').read()
```

可以使用 `lock()` 和 `unlock()` 方法避免同时访问文件可能导致的问题，`flush()` 强制将修改写入磁盘。

```
$ python mailbox_mbox_remove.py
Removing: 1
From MAILER-DAEMON Mon Nov 29 02:00:11 2010
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 1

This is the body.
>From (should be escaped).
There are 3 lines.
```

13.4.2 Maildir

创建 Maildir 格式是为了消除 mbox 文件并发修改存在的问题。这里不再使用单个文件，Maildir 邮箱会组织为一个目录，其中各个消息分别包含在自己单独的文件中。这样还允许邮箱

嵌套，所以可以扩展 Maildir 邮箱的 API，添加一些方法来处理子文件夹。

创建 Maildir 邮箱

创建 Maildir 邮箱和 mbox 邮箱很类似，惟一的区别是构造函数的参数是一个目录名而不是文件名。与前面一样，如果邮箱不存在，会在具体添加消息时创建。

```
import mailbox
import email.utils
import os

from_addr = email.utils.formataddr(('Author',
                                     'author@example.com'))
to_addr = email.utils.formataddr(('Recipient',
                                   'recipient@example.com'))

mbox = mailbox.Maildir('Example')
mbox.lock()
try:
    msg = mailbox.mboxMessage()
    msg.set_unixfrom('author Sat Feb 7 01:05:34 2009')
    msg['From'] = from_addr
    msg['To'] = to_addr
    msg['Subject'] = 'Sample message 1'
    msg.set_payload('\n'.join(['This is the body.',
                                'From (will not be escaped).',
                                'There are 3 lines.\n',
                                ]))

    mbox.add(msg)
    mbox.flush()

    msg = mailbox.mboxMessage()
    msg.set_unixfrom('author Sat Feb 7 01:05:34 2009')
    msg['From'] = from_addr
    msg['To'] = to_addr
    msg['Subject'] = 'Sample message 2'
    msg.set_payload('This is the second body.\n')
    mbox.add(msg)
    mbox.flush()
finally:
    mbox.unlock()

for dirname, subdirs, files in os.walk('Example'):
    print dirname
    print '\tDirectories:', subdirs
    for name in files:
        fullname = os.path.join(dirname, name)
        print
```

```

print '***', fullname
print open(fullname).read()
print '*' * 20

```

消息添加到邮箱时，它们会放在 `new` 子目录中。读取之后，客户程序可能会把它们移动到 `cur` 子目录。

警告： 尽管从多个进程写同一个 Maildir 是安全的，但 `add()` 不是线程安全的。需要使用信号量或其他锁定机制，避免同一个进程的多个线程同时修改邮箱。

```
$ python mailbox_maildir_create.py
```

Example

```
Directories: ['cur', 'new', 'tmp']
```

Example/cur

```
Directories: []
```

Example/new

```
Directories: []
```

```
*** Example/new/1290996011.M658966P16077Q1.farnsworth.local
```

```
From: Author <author@example.com>
```

```
To: Recipient <recipient@example.com>
```

```
Subject: Sample message 1
```

```
This is the body.
```

```
From (will not be escaped).
```

```
There are 3 lines.
```

```
*****
```

```
*** Example/new/1290996011.M660614P16077Q2.farnsworth.local
```

```
From: Author <author@example.com>
```

```
To: Recipient <recipient@example.com>
```

```
Subject: Sample message 2
```

```
This is the second body.
```

```
*****
```

Example/tmp

```
Directories: []
```

读取 Maildir 邮箱

读取一个已有的 Maildir 邮箱与读取 mbox 邮箱很类似。

```
import mailbox
```

```
mbox = mailbox.Maildir('Example')
```

```
for message in mbox:
```



```
print message['subject']
```

不能保证以某种特定的顺序读取消息。

```
$ python mailbox_maildir_read.py
```

```
Sample message 1
```

```
Sample message 2
```

从 Maildir 邮箱删除消息

要从一个 Maildir 邮箱删除已有的消息，可以将消息的键传递到 `remove()` 或者使用 `del`。

```
import mailbox
import os

mbox = mailbox.Maildir('Example')
mbox.lock()
try:
    to_remove = []
    for key, msg in mbox.iteritems():
        if '2' in msg['subject']:
            print 'Removing:', key
            to_remove.append(key)
    for key in to_remove:
        mbox.remove(key)
finally:
    mbox.flush()
    mbox.close()

for dirname, subdirs, files in os.walk('Example'):
    print dirname
    print '\tDirectories:', subdirs
    for name in files:
        fullname = os.path.join(dirname, name)
        print
        print '***', fullname
        print open(fullname).read()
        print '*' * 20
```

没有办法计算一个消息的键，所以应当使用 `iteritems()` 从邮箱同时获取键和消息对象。

```
$ python mailbox_maildir_remove.py
```

```
Removing: 1290996011.M660614P16077Q2.farnsworth.local
Example
```

```
Directories: ['cur', 'new', 'tmp']
```

```
Example/cur
```

```
Directories: []
```

```
Example/new
```

```
Directories: []
```

```
*** Example/new/1290996011.M658966P16077Q1.farnsworth.local
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 1
```

```
This is the body.
From (will not be escaped).
There are 3 lines.
```

```
*****
Example/tmp
    Directories: []
```

Maildir 文件夹

Maildir 邮箱的子目录或文件夹 (folder) 可以通过 Maildir 类的方法直接管理。调用者可以列出、获取、创建和删除一个给定邮箱的子目录。

```
import mailbox
import os

def show_maildir(name):
    os.system('find %s -print' % name)

mbox = mailbox.Maildir('Example')
print 'Before:', mbox.list_folders()
show_maildir('Example')

print
print '#' * 30
print

mbox.add_folder('subfolder')
print 'subfolder created:', mbox.list_folders()
show_maildir('Example')

subfolder = mbox.get_folder('subfolder')
print 'subfolder contents:', subfolder.list_folders()

print
print '#' * 30
print

subfolder.add_folder('second_level')
print 'second_level created:', subfolder.list_folders()
show_maildir('Example')
print
print '#' * 30
```



print

```
subfolder.remove_folder('second_level')
print 'second_level removed:', subfolder.list_folders()
show_maildir('Example')
```

构造文件夹的目录名时，要在文件夹名前面加一个点号（.）作为前缀。

```
$ python mailbox_maildir_folders.py
```

```
Example
Example/cur
Example/new
Example/new/1290996011.M658966P16077Q1.farnsworth.local
Example/tmp
Example
Example/.subfolder
Example/.subfolder/cur
Example/.subfolder/maildirfolder
Example/.subfolder/new
Example/.subfolder/tmp
Example/cur
Example/new
Example/new/1290996011.M658966P16077Q1.farnsworth.local
Example/tmp
Example
Example/.subfolder
Example/.subfolder/.second_level
Example/.subfolder/.second_level/cur
Example/.subfolder/.second_level/maildirfolder
Example/.subfolder/.second_level/new
Example/.subfolder/.second_level/tmp
Example/.subfolder/cur
Example/.subfolder/maildirfolder
Example/.subfolder/new
Example/.subfolder/tmp
Example/cur
Example/new
Example/new/1290996011.M658966P16077Q1.farnsworth.local
Example/tmp
Example
Example/.subfolder
Example/.subfolder/cur
Example/.subfolder/maildirfolder
Example/.subfolder/new
Example/.subfolder/tmp
Example/cur
Example/new
```



```
Example/new/1290996011.M658966P16077Q1.farnsworth.local
Example/tmp
Before: []

#####

subfolder created: ['subfolder']
subfolder contents: []

#####

second_level created: ['second_level']

#####

second_level removed: []
```

13.4.3 其他格式

mailbox 还支持另外一些格式，不过这些格式都没有 mbox 或 Maildir 常用。MH 也是一种多文件邮箱格式，一些邮箱处理程序就使用了这种格式。Babyl 和 MMDF 是单文件格式，使用了不同于 mbox 的消息分隔符。单文件格式支持的 API 与 mbox 相同，MH 则包括 Maildir 类中与文件夹相关的方法。

参见：

mailbox (<http://docs.python.org/library/mailbox.html>) 这个模块的标准库文档。

mbox manpage from qmail (<http://www.qmail.org/man/man5/mbox.html>) mbox 格式的文档。

Maildir manpage from qmail (<http://www.qmail.org/man/man5/maildir.html>) Maildir 格式的文档。

email email 模块。

mhlib mhlib 模块。

imaplib (13.3 节) imaplib 模块可以处理 IMAP 服务器上保存的邮件消息。



第 14 章

应用构建模块

Python 标准库的强大从它的规模就可见一斑。它包括了程序结构众多方面的实现，也正由于它提供的实现十分丰富，使得开发人员可以集中精力考虑如何让自己的应用别具一格，而不用反复实现所有这些基本的内容。本章将介绍一些更常重用的构建模块，它们可以解决大多数应用中常见的问题。

有 3 个不同模块可以用来解析不同样式的命令行参数。`getopt` 实现了 C 程序和 shell 脚本中同样的底层处理模型。与另外几个选项解析库相比，这个库的特性较少，不过由于其简单性和熟悉度，使之成为一个流行的选择。`optparse` 是一个更为现代、更为灵活的库，可以替代 `getopt`。`argparse` 是一个用于解析和验证命令行参数的第三方接口，它的出现使 `getopt` 和 `optparse` 都成为昨日黄花。它支持将参数从字符串转换为整数和其他类型，遇到某个选项时可以进行回调，可以为用户未提供的选项设置默认值，还可以为程序自动生成使用说明。

交互式程序应当使用 `readline` 为用户提供一个命令提示窗口。它包括一些工具，可以管理历史、自动完成命令，还可以用 `emacs` 和 `vi` 按键绑定交互式地完成编辑。为了安全地提示用户输入一个密码或其他秘密值，而不要在其键入时将值回显在屏幕上，可以使用 `getpass`。

`cmd` 模块为交互式、命令驱动的 shell 程序提供了一个框架。它提供了主循环，并处理与用户的交互，所以应用只需要实现各个命令的处理回调。

`shlex` 是一个用于 shell 语法的解析器，采用这种语法时，各行由 token 构成，并用空白符分隔。这个解析器足够“聪明”，可以很好地处理引号和转义序列，所以嵌入空格的文本会处理为单个 token。`shlex` 很适合作为领域特定语言的词法分析器，如配置文件或编程语言。

用 `ConfigParser` 可以很容易地管理应用配置文件。它能够在程序运行之间保存用户首选项，可以在下一次应用开始时读取用户首选项，甚至可以提供一个简单的数据文件格式。

真实世界中部署的应用要为用户提供调试信息。简单的错误消息和 `traceback` 会有帮助，不过如果很难再生问题，完整的活动日志可以直接指向导致失败的事件链。`logging` 模块包含一个功能完备的 API，可以管理日志文件、支持多个线程，甚至可以与远程日志守护进程交互来实现集中式日志。

对于 UNIX 环境中的程序，最常用的模式之一是逐行过滤器，即读取数据、修改数据，再将其写回。读取文件非常简单，不过要创建一个过滤器应用，再没有比使用 `fileinput` 模块更简单的方法了。其 API 是一个行迭代器，会提供各个输入行，所以程序主体是一个简单的 `for` 循环。这个模块会为要处理的文件名解析命令行参数，或者只是直接从标准输入读取，所以基于 `fileinput` 建立的工具可以在文件上直接运行，也可以作为某个管道的一部分。

解释器关闭一个程序时，可以使用 `atexit` 调度要运行的函数。注册退出回调对于释放资源（注销远程服务、关闭文件等等）很有用。

`sched` 模块实现了一个调度器，用于在将来某些时刻触发事件。这个 API 没有给出“时间”的定义，所以可以使用任何时间，从真实时钟时间到解释器步数都是允许的。

14.1 getopt——命令行选项解析

作用：命令行选项解析。

Python 版本：1.4 及以后版本

`getopt` 模块是原来的命令行选项解析器，支持 UNIX 函数 `getopt()` 建立的约定。它会解析一个参数序列，如 `sys.argv`，并返回一个元组序列和一个非选项参数序列，元组序列中包含一些（选项，参数）对。

目前支持的选项语法包括短格式和长格式选项：

```
-a
-bval
-b val
--noarg
--witharg=val
--witharg val
```

14.1.1 函数参数

`getopt()` 函数有 3 个参数：

- 第 1 个参数是要解析的参数序列。这通常来自 `sys.argv[1:]`（忽略 `sys.argv[0]` 中的程序名）。
- 第 2 个参数是单字符选项的选项定义串。如果某个选项需要一个参数，相应字母后面会有一个冒号。
- 第 3 个参数（如果使用）应当是一个长格式选项名序列。长格式选项可以包含多个字符，如 `--noarg` 或 `--witharg`。序列中的选项名不包括“--”前缀。如果某个长选项需要一个参数，其名应当有一个后缀“=”。

短格式和长格式选项可以在一个调用中结合使用。

14.1.2 短格式选项

这个示例程序接受 3 个选项。`-a` 是一个简单标志，`-b` 和 `-c` 需要一个参数。选项定义串为 `"ab:c:"`。

```
import getopt

opts, args = getopt.getopt(['-a', '-bval', '-c', 'val'], 'ab:c:')

for opt in opts:
    print opt
```


这个程序将一个模拟选项值列表传递到 `getopt()`，来显示如何对它们进行处理。

```
$ python getopt_short.py
```

```
('a', '')
('b', 'val')
('c', 'val')
```

14.1.3 长格式选项

对于一个有两个选项的程序 (`--noarg` 和 `--witharg`)，长参数序列为 `['noarg', 'witharg=']`。

```
import getopt
```

```
opts, args = getopt.getopt([ '--noarg',
                             '--witharg', 'val',
                             '--witharg2=another',
                             ],
                             '',
                             [ 'noarg', 'witharg=', 'witharg2=' ])
```

```
for opt in opts:
    print opt
```

由于这个示例程序没有任何短格式选项，所以 `getopt()` 的第 2 个参数是一个空串。

```
$ python getopt_long.py
```

```
('--noarg', '')
('--witharg', 'val')
('--witharg2', 'another')
```

14.1.4 一个完整的例子

这个例子是一个更完整的程序，它有 5 个选项：`-o`、`-v`、`--output`、`--verbose` 和 `--version`。`-o`、`--output` 和 `--version` 选项分别需要一个参数。

```
import getopt
import sys

version = '1.0'
verbose = False
output_filename = 'default.out'

print 'ARGV      : ', sys.argv[1:]

try:
    options, remainder = getopt.getopt(
        sys.argv[1:],
        'o:v',
        ['output=',
```



```

        'verbose',
        'version=',
    ])
except getopt.GetoptError as err:
    print 'ERROR:', err
    sys.exit(1)

print 'OPTIONS   :', options

for opt, arg in options:
    if opt in ('-o', '--output'):
        output_filename = arg
    elif opt in ('-v', '--verbose'):
        verbose = True
    elif opt == '--version':
        version = arg

print 'VERSION   :', version
print 'VERBOSE    :', verbose
print 'OUTPUT      :', output_filename
print 'REMAINING   :', remainder

```

可以采用多种不同方式调用这个程序。如果不带任何参数调用这个程序，会使用默认设置。

```
$ python getopt_example.py
```

```

ARGV      : []
OPTIONS    : []
VERSION    : 1.0
VERBOSE    : False
OUTPUT     : default.out
REMAINING  : []

```

单字母选项与其参数可以用空白符分隔。

```
$ python getopt_example.py -o foo
```

```

ARGV      : ['-o', 'foo']
OPTIONS    : [['-o', 'foo']]
VERSION    : 1.0
VERBOSE    : False
OUTPUT     : foo
REMAINING  : []

```

或者，也可以把选项和值结合到一个参数中。

```
$ python getopt_example.py -ofoo
```

```

ARGV      : ['-ofoo']
OPTIONS    : [['-o', 'foo']]

```



```

VERSION : 1.0
VERBOSE : False
OUTPUT  : foo
REMAINING : []

```

长格式选项也可以类似地与值分隔。

```
$ python getopt_example.py --output foo
```

```

ARGV      : ['--output', 'foo']
OPTIONS   : [['--output', 'foo']]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []

```

一个长选项与其值结合时，选项名和值要用一个 = 分隔。

```
$ python getopt_example.py --output=foo
```

```

ARGV      : ['--output=foo']
OPTIONS   : [['--output', 'foo']]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []

```

14.1.5 缩写长格式选项

只要提供了一个惟一的前缀，则不必在命令行上完整地拼写出长格式选项。

```
$ python getopt_example.py --o foo
```

```

ARGV      : ['--o', 'foo']
OPTIONS   : [['--output', 'foo']]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []

```

如果没有提供一个惟一的前缀，则会产生一个异常。

```
$ python getopt_example.py --ver 2.0
```

```

ARGV      : ['--ver', '2.0']
ERROR: option --ver not a unique prefix

```

14.1.6 GNU 选项解析

正常情况下，一旦遇到第一个非选项参数，选项处理就会停止。

```
$ python getopt_example.py -v not_an_option --output foo
```

```

ARGV      : ['-v', 'not_an_option', '--output', 'foo']
OPTIONS   : [['-v', '']]
VERSION   : 1.0
VERBOSE   : True
OUTPUT    : default.out
REMAINING : ['not_an_option', '--output', 'foo']

```

Python 2.3 中为这个模块增加了另外一个函数 `gnu_getopt()`。它允许在命令行上以任意顺序混合选项和非选项参数。

```

import getopt
import sys

version = '1.0'
verbose = False
output_filename = 'default.out'

print 'ARGV      :', sys.argv[1:]

try:
    options, remainder = getopt.gnu_getopt(
        sys.argv[1:],
        'o:v',
        ['output=',
         'verbose',
         'version='],
    )
except getopt.GetoptError as err:
    print 'ERROR:', err
    sys.exit(1)

print 'OPTIONS   :', options

for opt, arg in options:
    if opt in ('-o', '--output'):
        output_filename = arg
    elif opt in ('-v', '--verbose'):
        verbose = True
    elif opt == '--version':
        version = arg

print 'VERSION    :', version
print 'VERBOSE     :', verbose
print 'OUTPUT      :', output_filename
print 'REMAINING   :', remainder

```

修改前例中的调用之后，可以清楚地看出差别。



```
$ python getopt_gnu.py -v not_an_option --output foo

ARGV      : ['-v', 'not_an_option', '--output', 'foo']
OPTIONS   : [('-v', ''), ('--output', 'foo')]
VERSION   : 1.0
VERBOSE   : True
OUTPUT    : .foo
REMAINING : ['not_an_option']
```

14.1.7 结束参数处理

如果 `getopt()` 在输入参数中遇到 “--”，它会停止，不再将余下的参数作为选项处理。这个特性可以用来传递看上去像选项的参数值，如以一个短横线 (“-”) 开头的文件名。

```
$ python getopt_example.py -v -- --output foo

ARGV      : ['-v', '--', '--output', 'foo']
OPTIONS   : [('-v', '')]
VERSION   : 1.0
VERBOSE   : True
OUTPUT    : default.out
REMAINING : ['--output', 'foo']
```

参见：

`getopt` (<http://docs.python.org/library/getopt.html>) 这个模块的标准库文档。

`argparse` (14.3 节) `argparse` 模块取代了 `getopt` 和 `optparse`。

`optparse` (14.2 节) `optparse` 模块。

14.2 optparse——命令行选项解析器

作用：取代 `getopt` 的命令行选项解析器。

Python 版本：2.3 及以后版本

`optparse` 是当前替代 `getopt` 的模块，用来完成命令行选项解析，它可以提供 `getopt` 所没有的很多特性，包括类型转换、选项回调，以及自动生成帮助。除了这里能介绍的特性外，`optparse` 还有很多其他特性，不过本节会介绍较为常用的一些功能。

14.2.1 创建 OptionParser

用 `optparse` 解析选项有两个阶段。首先，构造 `OptionParser` 实例，并配置期望的选项。然后输入一个选项序列进行处理。

```
import optparse
parser = optparse.OptionParser()
```

一般地，一旦创建了解析器，要显式地向解析器添加各个选项，并提供信息指出命令行上

遇到该选项时该如何处理。也可以向 `OptionParser` 构造函数传递一个选项列表，不过这种做法不常用。

定义选项

应当使用 `add_option()` 方法一次增加一个选项。参数列表前面所有未命名的串参数都处理为选项名。要为一个选项创建别名（也就是同一个选项同时有短格式和长格式），可以传入多个名。

解析命令行

定义所有选项之后，可以向 `parse_args()` 传递一个参数串序列来解析命令行。默认情况下，参数由 `sys.argv[1:]` 得到，不过也可以显式地传递一个列表。这些选项使用 GNU/POSIX 语法处理，所以选项和参数值可以混合出现在序列中。

`parse_args()` 的返回值是一个两部分的元组，其中包含一个 `Values` 实例以及未解释为选项的命令参数列表。选项的默认处理动作是使用 `add_option()` 的 `dest` 参数中指定的名称存储这个值。`parse_args()` 返回的 `Values` 实例包含选项值（作为属性），所以，如果一个选项的 `dest` 设置为“`myoption`”，就可以作为 `options.myoption` 访问这个选项值。

14.2.2 短格式和长格式选项

下面是一个简单的例子，有 3 个不同选项：一个布尔选项（-a），一个简单的串选项（-b），以及一个整数选项（-c）。

```
import optparse

parser = optparse.OptionParser()
parser.add_option('-a', action="store_true", default=False)
parser.add_option('-b', action="store", dest="b")
parser.add_option('-c', action="store", dest="c", type="int")

print parser.parse_args(['-a', '-bval', '-c', '3'])
```

类似于 `getopt.gnu_getopt()`，`optparse` 用同样的规则解析命令行选项，所以有两种方法向单字符选项传值。这两种形式在例子中都用到，具体为 `-bval` 和 `-c val`。

```
$ python optparse_short.py

(<Values at 0x100e1b560: {'a': True, 'c': 3, 'b': 'val'}>, [])
```

与输出中“c”关联的值类型是一个整数，因为要求 `OptionParser` 存储参数之前先进行转换。不同于 `getopt`，`optparse` 对“长”选项名的处理并无不同。

```
import optparse

parser = optparse.OptionParser()
parser.add_option('--noarg', action="store_true", default=False)
```

```

parser.add_option('--witharg', action="store", dest="witharg")
parser.add_option('--witharg2', action="store",
                  dest="witharg2", type="int")

print parser.parse_args([ '--noarg',
                          '--witharg', 'val',
                          '--witharg2=3' ])

```

结果也是相似的。

```
$ python optparse_long.py
```

```

(<Values at 0x100e1b5a8: {'noarg': True, 'witharg': 'val',
'witharg2': 3}>, [])

```

14.2.3 用 getopt 比较

由于 optparse 要替换 getopt，下面这个例子重新实现了 getopt 一节中的示例程序。

```

import optparse
import sys

print 'ARGV      :', sys.argv[1:]

parser = optparse.OptionParser()
parser.add_option('-o', '--output',
                  dest="output_filename",
                  default="default.out",
                  )
parser.add_option('-v', '--verbose',
                  dest="verbose",
                  default=False,
                  action="store_true",
                  )
parser.add_option('--version',
                  dest="version",
                  default=1.0,
                  type="float",
                  )
options, remainder = parser.parse_args()

print 'VERSION   :', options.version
print 'VERBOSE    :', options.verbose
print 'OUTPUT      :', options.output_filename
print 'REMAINING   :', remainder

```

由于同时增加了选项 -o 和 --output，所以它们是别名。可以在命令行上使用其中任意一个选项。

```
$ python optparse_getoptcomparison.py -o output.txt
```

```
ARGV      : ['-o', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
```

```
$ python optparse_getoptcomparison.py --output output.txt
```

```
ARGV      : ['--output', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
```

还可以使用长选项的惟一前缀。

```
$ python optparse_getoptcomparison.py --out output.txt
```

```
ARGV      : ['--out', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
```

14.2.4 选项值

默认的处理动作是存储选项的参数。如果定义选项时提供了一个类型，在存储这个参数值之前会首先将该值转换为该类型。

设置默认值

根据定义，选项是可选的，所以如果命令行上未指定选项，应用应当建立默认行为。定义选项时可以使用参数 `default` 提供各选项的默认值。

```
import optparse

parser = optparse.OptionParser()
parser.add_option('-o', action="store", default="default value")

options, args = parser.parse_args()

print options.o
```

默认值要与期望的选项类型匹配，因为默认值不会完成任何转换。

```
$ python optparse_default.py
```

```
default value
```



```
$ python optparse_default.py -o "different value"
```

```
different value
```

还可以在定义选项之后使用 `set_defaults()` 的关键字参数加载默认值。

```
import optparse

parser = optparse.OptionParser()
parser.add_option('-o', action="store")

parser.set_defaults(o='default value')

options, args = parser.parse_args()

print options.o
```

如果从一个配置文件或其他来源加载默认值，而不是硬编码写入默认值，这种形式就很有用。

```
$ python optparse_set_defaults.py
```

```
default value
```

```
$ python optparse_set_defaults.py -o "different value"
```

```
different value
```

定义的所有选项都可以作为 `parse_args()` 返回的 `Values` 实例的属性得到，所以应用在尝试使用一个选项值之前不需要检查这个选项是否存在。

```
import optparse

parser = optparse.OptionParser()
parser.add_option('-o', action="store")

options, args = parser.parse_args()

print options.o
```

如果未向一个选项提供默认值，而且命令行上没有指定这个选项，它的值则为 `None`。

```
$ python optparse_no_default.py
```

```
None
```

```
$ python optparse_no_default.py -o "different value"
```

```
different value
```

类型转换

`optparse` 可以把选项值从字符串转换为整数、浮点数、长整数和复杂值。要启用转换，需要指定选项类型作为 `add_option()` 的参数。

```
import optparse
```

```
parser = optparse.OptionParser()
parser.add_option('-i', action="store", type="int")
parser.add_option('-f', action="store", type="float")
parser.add_option('-l', action="store", type="long")
parser.add_option('-c', action="store", type="complex")
```

```
options, args = parser.parse_args()
```

```
print 'int      : %-16r %s' % (type(options.i), options.i)
print 'float    : %-16r %s' % (type(options.f), options.f)
print 'long     : %-16r %s' % (type(options.l), options.l)
print 'complex: %-16r %s' % (type(options.c), options.c)
```

如果一个选项的值不能转换为指定的类型，会打印一个错误，程序将退出。

```
$ python optparse_types.py -i 1 -f 3.14 -l 1000000 -c 1+2j
```

```
int      : <type 'int'>      1
float    : <type 'float'>    3.14
long     : <type 'long'>     1000000
complex: <type 'complex'> (1+2j)
```

```
$ python optparse_types.py -i a
```

```
Usage: optparse_types.py [options]
```

```
optparse_types.py: error: option -i: invalid integer value: 'a'
```

通过派生 Option 类，可以创建定制转换。有关的更多详细信息请参考标准库文档。

枚举

choice 类型使用一个候选串列表来提供验证。将 type 设置为 choice，并使用 add_option() 的 choices 参数提供合法值列表。

```
import optparse
```

```
parser = optparse.OptionParser()
```

```
parser.add_option('-c', type='choice', choices=['a', 'b', 'c'])
```

```
options, args = parser.parse_args()
```

```
print 'Choice:', options.c
```

如果输入不合法会导致一个错误消息，显示允许的值列表。

```
$ python optparse_choice.py -c a
```

```

Choice: a

$ python optparse_choice.py -c b

Choice: b

$ python optparse_choice.py -c d

Usage: optparse_choice.py [options]

optparse_choice.py: error: option -c: invalid choice: 'd' (choose
from 'a', 'b', 'c')

```

14.2.5 选项动作

不同于 `getopt`（它只解析选项），`optparse` 则是一个选项处理库。选项可能触发不同的动作，由 `add_option()` 的 `action` 参数指定。支持的动作包括存储参数（单独存储，或者作为列表的一部分存储），遇到这个选项时存储一个常量值（包括对 Boolean 分支语句的 `true/false` 值的特殊处理），统计遇到一个选项的次数，以及调用一个回调。默认动作是 `store`，这不需要显式指定。

常量

选项表示一组固定的选择（如一个应用的操作模式）时，通过创建单独的显式选项，可以更容易地建立文档。`store_const` 动作就是要达到这个目的。

```

import optparse

parser = optparse.OptionParser()
parser.add_option('--earth', action="store_const",
                  const='earth', dest='element',
                  default='earth',
                  )
parser.add_option('--air', action='store_const',
                  const='air', dest='element',
                  )
parser.add_option('--water', action='store_const',
                  const='water', dest='element',
                  )
parser.add_option('--fire', action='store_const',
                  const='fire', dest='element',
                  )

options, args = parser.parse_args()

print options.element

```

`store_const` 动作将应用中的一个常量值与用户指定的选项相关联。可以配置多个选项对相同的 `dest` 名存储不同的常量值，这样一来，应用只需检查一个设置。

```
$ python optparse_store_const.py  
  
earth  
  
$ python optparse_store_const.py --fire  
  
fire
```

布尔标志

布尔选项使用特殊动作来实现，用来存储 true 和 false 常量值。

```
import optparse  
  
parser = optparse.OptionParser()  
parser.add_option('-t', action='store_true',  
                  default=False, dest='flag')  
parser.add_option('-f', action='store_false',  
                  default=False, dest='flag')  
  
options, args = parser.parse_args()  
  
print 'Flag:', options.flag
```

通过将 dest 名配置为相同的值，可以创建同一个标志的 true 和 false 版本。

```
$ python optparse_boolean.py  
  
Flag: False  
  
$ python optparse_boolean.py -t  
  
Flag: True  
  
$ python optparse_boolean.py -f  
  
Flag: False
```

重复选项

处理重复的选项有 3 种方法：覆盖、追加和统计。默认做法是覆盖所有现有的值，所以会使用最后指定的选项。store 动作就采用这种方式。

通过使用 append 动作，可以在选项重复时累积值，创建一个值列表。允许多个响应时，追加模式会很有用，因为各个响应可以单独列出。

```
import optparse  
  
parser = optparse.OptionParser()  
parser.add_option('-o', action="append", dest='outputs', default=[])  
options, args = parser.parse_args()
```

```
print options.outputs
```

命令行上指定值的顺序会保留，因为这个顺序可能对应用很重要。

```
$ python optparse_append.py
```

```
[]
```

```
$ python optparse_append.py -o a.out
```

```
['a.out']
```

```
$ python optparse_append.py -o a.out -o b.out
```

```
['a.out', 'b.out']
```

有时，知道一个选项指定了多少次就足够了，而不需要相关的值。例如，很多应用允许用户重复 `-v` 选项，来提高其输出的详细等级。每次这个选项出现时，`count` 动作会让一个值递增。

```
import optparse
```

```
parser = optparse.OptionParser()
parser.add_option('-v', action="count",
                  dest='verbosity', default=1)
parser.add_option('-q', action='store_const',
                  const=0, dest='verbosity')
```

```
options, args = parser.parse_args()
```

```
print options.verbosity
```

由于 `-v` 选项不带任何参数，可以使用语法 `-vv` 重复，也可以重复单个选项。

```
$ python optparse_count.py
```

```
1
```

```
$ python optparse_count.py -v
```

```
2
```

```
$ python optparse_count.py -v -v
```

```
3
```

```
$ python optparse_count.py -vv
```

```
3
```

```
$ python optparse_count.py -q
```



0

回调

除了直接保存选项参数，还可以为选项定义回调函数，在命令行上遇到选项时就会调用这些回调函数。选项的回调函数有 4 个参数：导致回调的 Option 实例、来自命令行的选项串、与选项关联的参数值，以及完成解析工作的 OptionParser 实例。

```
import optparse

def flag_callback(option, opt_str, value, parser):
    print 'flag_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

def with_callback(option, opt_str, value, parser):
    print 'with_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

parser = optparse.OptionParser()
parser.add_option('--flag', action="callback",
                  callback=flag_callback)
parser.add_option('--with',
                  action="callback",
                  callback=with_callback,
                  type="string",
                  help="Include optional feature")

parser.parse_args(['--with', 'foo', '--flag'])
```

在这个例子中，--with 选项配置为有一个字符串参数（也支持其他类型，如整数和浮点数）。

```
$ python optparse_callback.py
```

```
with_callback:
    option: <Option at 0x100e1b3b0: --with>
    opt_str: --with
    value: foo
    parser: <optparse.OptionParser instance at 0x100da1200>
flag_callback:
    option: <Option at 0x100e1b320: --flag>
    opt_str: --flag
    value: None
    parser: <optparse.OptionParser instance at 0x100da1200>
```

可以使用 `nargs` 选项配置回调有多个参数。

```
import optparse

def with_callback(option, opt_str, value, parser):
    print 'with_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

parser = optparse.OptionParser()
parser.add_option('--with',
                  action="callback",
                  callback=with_callback,
                  type="string",
                  nargs=2,
                  help="Include optional feature")

parser.parse_args(['--with', 'foo', 'bar'])
```

在这里，参数会通过 `value` 参数作为一个 `tuple` 传递给回调函数。

```
$ python optparse_callback_nargs.py
```

```
with_callback:
  option: <Option at 0x100e1a2d8: --with>
  opt_str: --with
  value: ('foo', 'bar')
  parser: <optparse.OptionParser instance at 0x100da0128>
```

14.2.6 帮助消息

`OptionParser` 会为所有选项集自动添加一个帮助选项，使用户可以在命令行上键入 `--help` 查看运行程序的使用说明。这个帮助消息包括所有选项，并指示它们是否有参数。还可以把帮助文本传入 `add_option()`，得到一个选项更详细的描述。

```
import optparse

parser = optparse.OptionParser()
parser.add_option('--no-foo', action="store_true",
                  default=False,
                  dest="foo",
                  help="Turn off foo",
                  )
parser.add_option('--with', action="store",
                  help="Include optional feature")
```

```
parser.parse_args()
```

选项按字母表顺序列出，并在同一行上给出别名。选项有参数时，会在帮助输出中包含 `dest` 名作为参数名。帮助文本打印在右列。

```
$ python optparse_help.py --help
```

```
Usage: optparse_help.py [options]
```

```
Options:
```

```
-h, --help      show this help message and exit
--no-foo        Turn off foo
--with=WITH     Include optional feature
```

随选项 `--with` 打印的 `WITH` 来自选项的目标变量。如果内部变量名没有提供足够的描述性，不适合放在文档中，可以使用 `metavar` 参数设置一个不同的变量名。

```
import optparse
```

```
parser = optparse.OptionParser()
parser.add_option('--no-foo', action="store_true",
                  default=False,
                  dest="foo",
                  help="Turn off foo",
                  )
parser.add_option('--with', action="store",
                  help="Include optional feature",
                  metavar='feature_NAME')
```

```
parser.parse_args()
```

值会原样打印，不做任何改变（如置为大写或添加标点符号）。

```
$ python optparse_metavar.py -h
```

```
Usage: optparse_metavar.py [options]
```

```
Options:
```

```
-h, --help      show this help message and exit
--no-foo        Turn off foo
--with=feature_NAME Include optional feature
```

组织选项

很多应用包括一些相关选项集合。例如，`rpm` 对于各种操作模式有单独的选项。`optparse` 使用选项组（option groups）在帮助输出中组织选项。选项值仍然保存在一个 `Values` 实例中，所以选项名的命名空间仍是平面命名空间[⊖]。

⊖ 平面命名空间（flat namespace）是一种非结构化的命名空间。在平面命名空间中，每个对象必须有惟一的名称。——译者注


```

import optparse

parser = optparse.OptionParser()
parser.add_option('-q', action='store_const',
                  const='query', dest='mode',
                  help='Query')
parser.add_option('-i', action='store_const',
                  const='install', dest='mode',
                  help='Install')

query_opts = optparse.OptionGroup(
    parser, 'Query Options',
    'These options control the query mode.',
)
query_opts.add_option('-l', action='store_const',
                      const='list', dest='query_mode',
                      help='List contents')
query_opts.add_option('-f', action='store_const',
                      const='file', dest='query_mode',
                      help='Show owner of file')
query_opts.add_option('-a', action='store_const',
                      const='all', dest='query_mode',
                      help='Show all packages')
parser.add_option_group(query_opts)

install_opts = optparse.OptionGroup(
    parser, 'Installation Options',
    'These options control installation.',
)
install_opts.add_option(
    '--hash', action='store_true', default=False,
    help='Show hash marks as progress indication')
install_opts.add_option(
    '--force', dest='install_force', action='store_true',
    default=False,
    help='Install, regardless of dependencies or existing version')
parser.add_option_group(install_opts)

```

```
print parser.parse_args()
```

每组分别有自己的节标题和描述，选项会显示在一起。

```
$ python optparse_groups.py -h
```

```
Usage: optparse_groups.py [options]
```

Options:

```

-h, --help  show this help message and exit
-q          Query

```

```
-i          Install
```

Query Options:

These options control the query mode.

```
-l          List contents
-f          Show owner of file
-a          Show all packages
```

Installation Options:

These options control installation.

```
--hash     Show hash marks as progress indication
--force     Install, regardless of dependencies or existing version
```

应用设置

自动生成帮助的工具使用配置设置来控制帮助输出的很多方面。程序的用法 (usage) 串会显示如何指定位置参数, 可以在创建 `OptionParser` 时设置这个串。

```
import optparse
```

```
parser = optparse.OptionParser(
    usage='%prog [options] <arg1> <arg2> [<arg3>...]'
)
parser.add_option('-a', action="store_true", default=False)
parser.add_option('-b', action="store", dest="b")
parser.add_option('-c', action="store", dest="c", type="int")

parser.parse_args()
```

字面量值 `%prog` 在运行时会扩展为程序名, 从而反映脚本的完整路径。如果脚本由 `python` 运行, 而不是直接运行, 则使用脚本名。

```
$ python optparse_usage.py -h
```

```
Usage: optparse_usage.py [options] <arg1> <arg2> [<arg3>...]
```

Options:

```
-h, --help  show this help message and exit
-a
-b B
-c C
```

可以使用 `prog` 参数改变程序名。

```
import optparse
```

```
parser = optparse.OptionParser(
    usage='%prog [options] <arg1> <arg2> [<arg3>...]',
```



```

    prog='my_program_name',
)
parser.add_option('-a', action="store_true", default=False)
parser.add_option('-b', action="store", dest="b")
parser.add_option('-c', action="store", dest="c", type="int")

parser.parse_args()

```

不过，以这种方式硬编码写入程序名通常不是一个好想法，因为如果对程序重命名，帮助将不会反映这个变化。

```
$ python optparse_prog.py -h
```

```
Usage: my_program_name [options] <arg1> <arg2> [<arg3>...]
```

```
Options:
```

```

-h, --help  show this help message and exit
-a
-b B
-c C

```

可以使用 `version` 参数设置应用版本。提供一个版本值时，`optparse` 会向解析器自动添加一个 `--version` 选项。

```
import optparse
```

```

parser = optparse.OptionParser(
    usage='%prog [options] <arg1> <arg2> [<arg3>...]',
    version='1.0',
)

```

```
parser.parse_args()
```

用户运行程序时如果提供 `--version` 选项，`optparse` 会打印出这个版本串，然后退出。

```
$ python optparse_version.py -h
```

```
Usage: optparse_version.py [options] <arg1> <arg2> [<arg3>...]
```

```
Options:
```

```

--version  show program's version number and exit
-h, --help  show this help message and exit

```

```
$ python optparse_version.py --version
```

```
1.0
```

参见：

`optparse` (<http://docs.python.org/lib/module-optparse.html>) 这个模块的标准库文档。

getopt (14.1 节) getopt 模块, 被 optparse 取代。

argparse (14.3 节) 替代 optparse 的更新模块。

14.3 argparse——命令行选项和参数解析

作用: 命令行选项和参数解析。

Python 版本: 2.7 及以后版本

Python 2.7 中增加了 argparse 模块, 以取代 optparse。argparse 的实现支持一些新特性, 这些特性有些不能很容易地增加到 optparse 中, 有些则要求不能保证向后兼容的 API 改变。所以, 干脆在库中引入了一个新的模块。现在仍支持 optparse, 不过不太可能增加新特性了。

14.3.1 与 optparse 比较

argparse 与 optparse 提供的 API 很类似, 很多情况下, argparse 模块可以直接替代 optparse, 只需更新所用的类和方法名。不过, 还有一些情况下, 由于增加了新特性, 所以不能保留直接兼容性。

要根据具体情况来决定是否对现有程序升级。如果一个应用包含额外的代码来避开 optparse 的限制, 升级则可以减少维护工作。对于一个新程序, 如果将要部署这个程序的所有平台上都可以得到 argparse, 就应当对这个新程序使用 argparse。

14.3.2 建立解析器

使用 argparse 的第一步是创建一个解析器对象, 并告诉它需要什么参数。程序运行时可以使用这个解析器处理命令行参数。解析器类 (ArgumentParser) 的构造函数可以取多个参数, 为程序建立帮助文本中使用的描述以及其他全局行为或设置。

```
import argparse
parser = argparse.ArgumentParser(
    description='This is a PyMOTW sample program',
)
```

14.3.3 定义参数

argparse 是一个完整的参数处理库。参数可以触发不同的动作, 由 add_argument() 的 action 参数指定。支持的动作包括存储参数 (单独存储, 或者作为列表的一部分存储)、遇到这个参数时存储一个常量值 (包括对 Boolean 分支语句的 true/false 值的特殊处理)、统计遇到一个参数的次数, 以及调用一个回调来使用定制处理指令。

默认动作是存储参数值。如果提供了一个类型, 存储值之前要将值转换为该类型。如果提供了 dest 参数, 解析命令行参数时要用这个名称来保存值。

14.3.4 解析命令行

定义了所有参数之后，可以将一个参数串序列传递到 `parse_args()` 来解析命令行。默认情况下，参数由 `sys.argv[1:]` 得到，不过也可以使用任意的串列表。选项使用 GNU/POSIX 语法处理，所以选项和参数值可以混合出现在序列中。

`parse_args()` 的返回值是一个包含命令参数的 `Namespace`。这个对象会保存参数值（作为属性），所以如果参数的 `dest` 设置为 “`myoption`”，就可以作为 `args.myoption` 访问这个值。

14.3.5 简单示例

下面是一个简单的例子，有 3 个不同选项：一个布尔选项 (`-a`)，一个简单的串选项 (`-b`)，以及一个整数选项 (`-c`)。

```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print parser.parse_args(['-a', '-bval', '-c', '3'])
```

向单字符选项传值有多种方法。前面的例子使用了两种不同形式，`-bval` 和 `-c val`。

```
$ python argparse_short.py
```

```
Namespace(a=True, b='val', c=3)
```

与输出中 “`c`” 关联的值类型是一个整数，因为要求 `ArgumentParser` 存储参数之前先完成转换。“长” 选项名（名称中包含多个字符）也用同样的方式处理。

```
import argparse

parser = argparse.ArgumentParser(
    description='Example with long option names',
)

parser.add_argument('--noarg', action="store_true",
                    default=False)
parser.add_argument('--witharg', action="store",
                    dest="witharg")
parser.add_argument('--witharg2', action="store",
                    dest="witharg2", type=int)

print parser.parse_args(
    [ '--noarg', '--witharg', 'val', '--witharg2=3' ]
)
```



结果也是类似的。

```
$ python argparse_long.py
```

```
Namespace(noarg=True, witharg='val', witharg2=3)
```

有一个方面 `argparse` 与 `optparse` 有所不同，即对非选项参数值的处理。`optparse` 只完成选项解析，`argparse` 则是一个完备的命令行参数解析工具，还可以处理非选项参数。

```
import argparse
```

```
parser = argparse.ArgumentParser(
    description='Example with nonoptional arguments',
)
```

```
parser.add_argument('count', action="store", type=int)
parser.add_argument('units', action="store")
```

```
print parser.parse_args()
```

在这个例子中，“count”参数是一个整数，“units”参数保存为一个字符串。如果命令行中遗漏其中任何一个参数，或者给定值不能转换为正确的类型，就会报告一个错误。

```
$ python argparse_arguments.py 3 inches
```

```
Namespace(count=3, units='inches')
```

```
$ python argparse_arguments.py some inches
```

```
usage: argparse_arguments.py [-h] count units
argparse_arguments.py: error: argument count: invalid int value:
'some'
```

```
$ python argparse_arguments.py
```

```
usage: argparse_arguments.py [-h] count units
argparse_arguments.py: error: too few arguments
```

参数动作

遇到一个参数时会触发 6 个内置动作。

store 保存值，可能首先要将值转换为一个不同的类型（可选）。如果没有显式指定任何动作，这将是默认动作。

store_const 保存作为参数规范的一部分定义的一个值，而不是来自所解析的参数。这通常用于实现非布尔类型的命令行标志。

store_true / store_false 保存适当的布尔值。这些动作用于实现 Boolean 分支语句。

append 将值保存到一个列表。如果参数重复则会保存多个值。

append_const 将参数规范中定义的一个值保存到一个列表。

version 打印程序的版本详细信息，然后退出。

下面这个示例程序展示了各种动作类型，这里提供了触发各个动作所需的最小配置。

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-s', action='store',
                    dest='simple_value',
                    help='Store a simple value')

parser.add_argument('-c', action='store_const',
                    dest='constant_value',
                    const='value-to-store',
                    help='Store a constant value')

parser.add_argument('-t', action='store_true',
                    default=False,
                    dest='boolean_switch',
                    help='Set a switch to true')
parser.add_argument('-f', action='store_false',
                    default=False,
                    dest='boolean_switch',
                    help='Set a switch to false')

parser.add_argument('-a', action='append',
                    dest='collection',
                    default=[],
                    help='Add repeated values to a list')

parser.add_argument('-A', action='append_const',
                    dest='const_collection',
                    const='value-1-to-append',
                    default=[],
                    help='Add different values to list')
parser.add_argument('-B', action='append_const',
                    dest='const_collection',
                    const='value-2-to-append',
                    help='Add different values to list')

parser.add_argument('--version', action='version',
                    version='%(prog)s 1.0')

results = parser.parse_args()
print 'simple_value      = %r' % results.simple_value
print 'constant_value   = %r' % results.constant_value
print 'boolean_switch   = %r' % results.boolean_switch
```

```
print 'collection      = %r' % results.collection
print 'const_collection = %r' % results.const_collection
```

-t 和 -f 选项配置为修改同一个选项值，所以它们相当于一个布尔开关。-A 和 -B 的 dest 值相同，因此其常量值会追加到同一个列表。

```
$ python argparse_action.py -h
```

```
usage: argparse_action.py [-h] [-s SIMPLE_VALUE] [-c] [-t] [-f]
                        [-a COLLECTION] [-A] [-B] [--version]
```

optional arguments:

```
-h, --help            show this help message and exit
-s SIMPLE_VALUE       Store a simple value
-c                   Store a constant value
-t                   Set a switch to true
-f                   Set a switch to false
-a COLLECTION         Add repeated values to a list
-A                   Add different values to list
-B                   Add different values to list
--version             show program's version number and exit
```

```
$ python argparse_action.py -s value
```

```
simple_value      = 'value'
constant_value   = None
boolean_switch   = False
collection       = []
const_collection = []
```

```
$ python argparse_action.py -c
```

```
simple_value      = None
constant_value    = 'value-to-store'
boolean_switch    = False
collection       = []
const_collection = []
```

```
$ python argparse_action.py -t
```

```
simple_value      = None
constant_value    = None
boolean_switch    = True
collection       = []
const_collection = []
```

```
$ python argparse_action.py -f
```



```

simple_value      = None
constant_value   = None
boolean_switch   = False
collection        = []
const_collection = []

$ python argparse_action.py -a one -a two -a three

simple_value      = None
constant_value   = None
boolean_switch   = False
collection        = ['one', 'two', 'three']
const_collection = []

$ python argparse_action.py -B -A

simple_value      = None
constant_value   = None
boolean_switch   = False
collection        = []
const_collection = ['value-2-to-append', 'value-1-to-append']

$ python argparse_action.py --version

argparse_action.py 1.0

```

选项前缀

选项的默认语法基于一个 UNIX 约定：使用一个短横线（“-”）前缀来指示命令行开关。`argparse` 还支持其他前缀，所以程序可以采用本地平台的默认设置（也就是说，在 Windows 上则使用 “/”），或者遵循一个不同的约定。

```

import argparse

parser = argparse.ArgumentParser(
    description='Change the option prefix characters',
    prefix_chars='-+/',
)

parser.add_argument('-a', action="store_false",
                    default=None,
                    help='Turn A off',
)
parser.add_argument('+a', action="store_true",
                    default=None,
                    help='Turn A on',
)
parser.add_argument('//noarg', '+noarg',

```

```

        action="store_true",
        default=False)

```

```

print parser.parse_args()

```

将 ArgumentParser 的 prefix_chars 参数设置为一个字符串，其中包含允许指示选项的所有字符。有一点要了解，尽管 prefix_chars 建立了允许的开关字符，但是要由单独的参数定义来指定一个给定开关的语法，这一点很重要。这样就能显式地控制使用不同前缀的选项究竟是别名（如平台独立的命令行语法就属于这种情况）还是替代选项（例如，使用 “+” 指示打开一个开关，用 “-” 关闭开关）。在前面的例子中，+a 和 -a 是不同的参数，//noarg 也可以指定为 ++noarg，不过不能指定为 --noarg。

```

$ python argparse_prefix_chars.py -h

```

```

usage: argparse_prefix_chars.py [-h] [-a] [+a] [//noarg]

```

```

Change the option prefix characters

```

```

optional arguments:

```

```

  -h, --help            show this help message and exit
  -a                    Turn A off
  +a                    Turn A on
  //noarg, ++noarg

```

```

$ python argparse_prefix_chars.py +a

```

```

Namespace(a=True, noarg=False)

```

```

$ python argparse_prefix_chars.py -a

```

```

Namespace(a=False, noarg=False)

```

```

$ python argparse_prefix_chars.py //noarg

```

```

Namespace(a=None, noarg=True)

```

```

$ python argparse_prefix_chars.py ++noarg

```

```

Namespace(a=None, noarg=True)

```

```

$ python argparse_prefix_chars.py --noarg

```

```

usage: argparse_prefix_chars.py [-h] [-a] [+a] [//noarg]

```

```

argparse_prefix_chars.py: error: unrecognized arguments: --noarg

```

参数来源

在目前为止的例子中，提供给解析器的参数列表都是显式传入的一个列表，或者隐式地从 `sys.argv` 取参数。有一些类命令行指令并非来自命令行（如配置文件中的指令），使用 `argparse` 处理这种指令时，显式地传入列表会很有用。

```
import argparse
from ConfigParser import ConfigParser
import shlex

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

config = ConfigParser()
config.read('argparse_with_shlex.ini')
config_value = config.get('cli', 'options')
print 'Config :', config_value

argument_list = shlex.split(config_value)
print 'Arg List:', argument_list

print 'Results :', parser.parse_args(argument_list)
```

利用 `shlex` 可以很容易地分解存储在配置文件中的字符串。

```
$ python argparse_with_shlex.py

Config : -a -b 2
Arg List: ['-a', '-b', '2']
Results : Namespace(a=True, b='2', c=None)
```

应用代码中处理配置文件还有一种做法，可以使用 `fromfile_prefix_chars` 告诉 `argparse` 如何识别一个指定输入文件的参数（这个文件中包含一组要处理的参数）。

```
import argparse
from ConfigParser import ConfigParser
import shlex

parser = argparse.ArgumentParser(description='Short sample app',
                                fromfile_prefix_chars='@',
                                )

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print parser.parse_args(['@argparse_fromfile_prefix_chars.txt'])
```

这个例子发现一个有 @ 前缀的参数时会停止，然后读取指定的文件来查找更多参数。例如，输入文件 `argparse_fromfile_prefix_chars.txt` 包含一系列参数，每个参数各占一行。

```
-a
-b
2
```

处理这个文件时生成的输出如下。

```
$ python argparse_fromfile_prefix_chars.py
```

```
Namespace(a=True, b='2', c=None)
```

14.3.6 自动生成的选项

`argparse` 会自动添加选项来生成帮助，并显示应用的版本信息（如果有这个配置）。

`ArgumentParser` 的 `add_help` 参数控制与帮助相关的选项。

```
import argparse
```

```
parser = argparse.ArgumentParser(add_help=True)
```

```
parser.add_argument('-a', action="store_true", default=False)
```

```
parser.add_argument('-b', action="store", dest="b")
```

```
parser.add_argument('-c', action="store", dest="c", type=int)
```

```
print parser.parse_args()
```

会默认增加帮助选项（`-h` 和 `--help`），不过也可以将 `add_help` 设置为 `false` 禁用这些帮助选项。

```
import argparse
```

```
parser = argparse.ArgumentParser(add_help=False)
```

```
parser.add_argument('-a', action="store_true", default=False)
```

```
parser.add_argument('-b', action="store", dest="b")
```

```
parser.add_argument('-c', action="store", dest="c", type=int)
```

```
print parser.parse_args()
```

尽管 `-h` 和 `--help` 是请求帮助的事实标准选项名，不过有些应用或 `argparse` 的某些用法可能不需要提供帮助，或者需要用这些选项名来提供其他用途。

```
$ python argparse_with_help.py -h
```

```
usage: argparse_with_help.py [-h] [-a] [-b B] [-c C]
```

```
optional arguments:
```

```
-h, --help show this help message and exit
```

```
-a
```

```
-b B
```

```

-c C

$ python argparse_without_help.py -h

usage: argparse_without_help.py [-a] [-b B] [-c C]
argparse_without_help.py: error: unrecognized arguments: -h

```

在 ArgumentParser 构造函数中设置版本 (version) 时会增加版本选项 (-v 和 --version)。

```

import argparse

parser = argparse.ArgumentParser(version='1.0')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

```

```

print parser.parse_args()

print 'This is not printed'

```

这两种格式的选项都会打印程序的版本串, 然后立即退出。

```

$ python argparse_with_version.py -h

usage: argparse_with_version.py [-h] [-v] [-a] [-b B] [-c C]

optional arguments:
  -h, --help            show this help message and exit
  -v, --version          show program's version number and exit
  -a
  -b B
  -c C

$ python argparse_with_version.py -v

1.0

$ python argparse_with_version.py --version

1.0

```

14.3.7 解析器组织

argparse 包含很多特性来组织参数解析器, 以便于实现, 或者改善帮助输出的可用性。

共享解析器原则

程序员通常要实现一组命令行工具, 它们都取一组参数, 然后以某种方式特殊化。例如, 如果程序在采取具体行动之前都需要认证用户, 它们就都需要支持 --user 和 --password 选项。

不必显式地将这些选项添加到每一个 `ArgumentParser` 中，完全可以用这些共享选项定义一个父解析器，然后让各个程序的解析器继承它的选项。

第一步是用共享参数定义来建立解析器。由于父解析器的各个后续用户会尝试添加相同的帮助选项，这会导致一个异常，所以要在基解析器中关闭自动帮助生成。

```
import argparse

parser = argparse.ArgumentParser(add_help=False)
parser.add_argument('--user', action="store")
parser.add_argument('--password', action="store")
```

接下来，用 `parents` 集合创建另一个解析器。

```
import argparse
import argparse_parent_base

parser = argparse.ArgumentParser(
    parents=[argparse_parent_base.parser],
)

parser.add_argument('--local-arg',
                    action="store_true",
                    default=False)

print parser.parse_args()
```

得到的程序将有 3 个选项。

```
$ python argparse_uses_parent.py -h

usage: argparse_uses_parent.py [-h] [--user USER]
                                [--password PASSWORD]
                                [--local-arg]

optional arguments:
  -h, --help            show this help message and exit
  --user USER
  --password PASSWORD
  --local-arg
```

选项冲突

前面的例子指出，使用相同的参数名向一个解析器添加两个参数处理程序时，会导致一个异常。可以传入一个 `conflict_handler` 改变冲突解决行为。有两个内置的处理程序，分别是 `error`（默认）和 `resolve`，它们会根据处理程序添加的顺序来选择处理程序。

```
import argparse

parser = argparse.ArgumentParser(conflict_handler='resolve')
```

```

parser.add_argument('-a', action="store")
parser.add_argument('-b', action="store", help='Short alone')
parser.add_argument('--long-b', '-b',
                    action="store",
                    help='Long and short together')

print parser.parse_args(['-h'])

```

在这个例子中，由于使用了给定参数名的最后一个处理程序，独立选项 -b 被 --long-b 的别名屏蔽。

```

$ python argparse_conflict_handler_resolve.py

usage: argparse_conflict_handler_resolve.py [-h] [-a A]
[--long-b LONG_B]

optional arguments:
  -h, --help            show this help message and exit
  -a A                  Long and short together
  --long-b LONG_B, -b LONG_B

```

改变 add_argument() 调用的顺序，就可以不再屏蔽独立选项。

```

import argparse

parser = argparse.ArgumentParser(conflict_handler='resolve')

parser.add_argument('-a', action="store")
parser.add_argument('--long-b', '-b',
                    action="store",
                    help='Long and short together')
parser.add_argument('-b', action="store", help='Short alone')

print parser.parse_args(['-h'])

```

现在两个选项可以一起使用。

```

$ python argparse_conflict_handler_resolve2.py

usage: argparse_conflict_handler_resolve2.py [-h] [-a A]
[--long-b LONG_B]
[-b B]

optional arguments:
  -h, --help            show this help message and exit
  -a A                  Long and short together
  --long-b LONG_B      Long and short together
  -b B                  Short alone

```

参数组

argparse 将参数定义合并为“组”。默认情况下，它会使用两个组，一个对应选项，另一个对应必要的基于位置的参数。

```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('--optional', action="store_true", default=False)
parser.add_argument('positional', action="store")

print parser.parse_args()
```

帮助输出的“位置参数”（positional argument）和“可选参数”（optional argument）部分可以反映出这种分组。

```
$ python argparse_default_grouping.py -h

usage: argparse_default_grouping.py [-h] [--optional] positional

Short sample app

positional arguments:
  positional

optional arguments:
  -h, --help  show this help message and exit
  --optional
```

可以调整这种分组，使帮助更有条理，将相关的选项或值放在一起。可以使用定制组重新编写前面的共享选项示例，使得认证选项一同出现在帮助中。

用 `add_argument_group()` 创建“authentication”组，然后将与认证有关的各个选项添加到这个组，而不是添加到基解析器。

```
import argparse

parser = argparse.ArgumentParser(add_help=False)

group = parser.add_argument_group('authentication')

group.add_argument('--user', action="store")
group.add_argument('--password', action="store")
```

如果程序使用了基于组的父解析器，要像前面一样把它列在 `parents` 值中。

```
import argparse
import argparse_parent_with_group
```




```
parser = argparse.ArgumentParser(
    parents=[argparse_parent_with_group.parser],
)
```

```
parser.add_argument('--local-arg',
                    action="store_true",
                    default=False)
```

```
print parser.parse_args()
```

现在帮助输出会把认证选项显示在一起。

```
$ python argparse_uses_parent_with_group.py -h
```

```
usage: argparse_uses_parent_with_group.py [-h] [--user USER]
                                           [--password PASSWORD]
                                           [--local-arg]
```

optional arguments:

```
-h, --help            show this help message and exit
--local-arg
```

authentication:

```
--user USER
--password PASSWORD
```

互斥选项

定义互斥选项是选项分组特性的一个特殊情况，要使用 `add_mutually_exclusive_group()` 而不是 `add_argument_group()`。

```
import argparse
```

```
parser = argparse.ArgumentParser()
```

```
group = parser.add_mutually_exclusive_group()
group.add_argument('-a', action='store_true')
group.add_argument('-b', action='store_true')
```

```
print parser.parse_args()
```

`argparse` 会保证这种互斥性，从而只能指定组中的一个选项。

```
$ python argparse_mutually_exclusive.py -h
```

```
usage: argparse_mutually_exclusive.py [-h] [-a | -b]
```

optional arguments:

```
-h, --help  show this help message and exit
-a
-b
```



```
$ python argparse_mutually_exclusive.py -a
Namespace(a=True, b=False)

$ python argparse_mutually_exclusive.py -b
Namespace(a=False, b=True)

$ python argparse_mutually_exclusive.py -a -b
usage: argparse_mutually_exclusive.py [-h] [-a | -b]
argparse_mutually_exclusive.py: error: argument -b: not allowed with
argument -a
```

嵌套解析器

前面介绍的父解析器方法只是在相关命令之间共享选项的一种方法。还可以采用另一种方法，将命令结合到一个程序中，使用子解析器来处理命令行的各个部分。其做法类似于 `svn`、`hg` 和其他有多个命令动作或子命令的程序。

如果一个程序要处理文件系统上的目录，可以定义命令来创建、删除和列出目录的内容，如下所示。

```
import argparse

parser = argparse.ArgumentParser()

subparsers = parser.add_subparsers(help='commands')

# A list command
list_parser = subparsers.add_parser(
    'list', help='List contents')
list_parser.add_argument(
    'dirname', action='store',
    help='Directory to list')

# A create command
create_parser = subparsers.add_parser(
    'create', help='Create a directory')
create_parser.add_argument(
    'dirname', action='store',
    help='New directory to create')
create_parser.add_argument(
    '--read-only', default=False, action='store_true',
    help='Set permissions to prevent writing to the directory',
)

# A delete command
```

```

delete_parser = subparsers.add_parser(
    'delete', help='Remove a directory')
delete_parser.add_argument(
    'dirname', action='store', help='The directory to remove')
delete_parser.add_argument(
    '--recursive', '-r', default=False, action='store_true',
    help='Remove the contents of the directory, too',
)

```

```
print parser.parse_args()
```

帮助输出显示子解析器名为“commands”，可以在命令行上作为位置参数指定这个子解析器。

```
$ python argparse_subparsers.py -h
```

```
usage: argparse_subparsers.py [-h] {create,list,delete} ...
```

```
positional arguments:
  {create,list,delete}  commands
  list                  List contents
  create                Create a directory
  delete                Remove a directory

```

```
optional arguments:
  -h, --help            show this help message and exit

```

每个子解析器还有自己的帮助，描述该命令的参数和选项。

```
$ python argparse_subparsers.py create -h
```

```
usage: argparse_subparsers.py create [-h] [--read-only] dirname
```

```
positional arguments:
  dirname          New directory to create

```

```
optional arguments:
  -h, --help        show this help message and exit
  --read-only       Set permissions to prevent writing to the directory

```

解析参数时，`parse_args()` 返回的 `Namespace` 对象只包含与指定命令相关的值。

```
$ python argparse_subparsers.py delete -r foo
```

```
Namespace(dirname='foo', recursive=True)
```

14.3.8 高级参数处理

到目前为止，前面的例子展示了简单的布尔标志、带字符串或数值参数的选项，以及位置参数。`argparse` 还支持变长参数表、枚举和常量值的复杂参数规范。

可变参数表

参数定义可以配置为要利用所解析的命令行上的多个参数。根据所需或期望的参数个数，可以将 `nargs` 设置为表 14.1 所示的某个标志值。

表 14.1 `argparse` 中的可变参数定义标志

值	含 义
N	参数的绝对个数（例如 3）
?	0 或 1 个参数
*	0 或所有参数
+	所有（至少 1 个）参数

```
import argparse
```

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('--three', nargs=3)
```

```
parser.add_argument('--optional', nargs='?')
```

```
parser.add_argument('--all', nargs='*', dest='all')
```

```
parser.add_argument('--one-or-more', nargs='+')
```

```
print parser.parse_args()
```

解析器会执行参数统计指令，并生成一个准确的语法图作为命令帮助文档的一部分。

```
$ python argparse_nargs.py -h
```

```
usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
                        [--optional [OPTIONAL]]
                        [--all [ALL [ALL ...]]]
                        [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]
```

```
optional arguments:
```

```
-h, --help            show this help message and exit
```

```
--three THREE THREE THREE
```

```
--optional [OPTIONAL]
```

```
--all [ALL [ALL ...]]
```

```
--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]
```

```
$ python argparse_nargs.py
```

```
Namespace(all=None, one_or_more=None, optional=None, three=None)
```

```
$ python argparse_nargs.py --three
```

```
usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
```

```

        [--optional [OPTIONAL]]
        [--all [ALL [ALL ...]]]
        [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]
argparse_nargs.py: error: argument --three: expected 3
argument(s)

$ python argparse_nargs.py --three a b c

Namespace(all=None, one_or_more=None, optional=None,
three=['a', 'b', 'c'])

$ python argparse_nargs.py --optional

Namespace(all=None, one_or_more=None, optional=None, three=None)

$ python argparse_nargs.py --optional with_value

Namespace(all=None, one_or_more=None, optional='with_value',
three=None)

$ python argparse_nargs.py --all with multiple values

Namespace(all=['with', 'multiple', 'values'], one_or_more=None,
optional=None, three=None)

$ python argparse_nargs.py --one-or-more with_value

Namespace(all=None, one_or_more=['with_value'], optional=None,
three=None)

$ python argparse_nargs.py --one-or-more with multiple values
Namespace(all=None, one_or_more=['with', 'multiple', 'values'],
optional=None, three=None)

$ python argparse_nargs.py --one-or-more

usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
        [--optional [OPTIONAL]]
        [--all [ALL [ALL ...]]]
        [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]
argparse_nargs.py: error: argument --one-or-more: expected
at least one argument

```

参数类型

`argparse` 将所有参数值都处理为字符串，除非明确要求将字符串转换为另一个类型。`add_argument()` 的 `type` 参数定义了一个转换器函数，`ArgumentParser` 使用这个函数将参数值从字符

串转换为另外一种类型。

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-i', type=int)
parser.add_argument('-f', type=float)
parser.add_argument('--file', type=file)

try:
    print parser.parse_args()
except IOError, msg:
    parser.error(str(msg))
```

所有取一个字符串参数的可调用对象 (callable) 都可以作为 type 参数传入, 包括内置类型, 如 int()、float() 和 file()。

```
$ python argparse_type.py -i 1

Namespace(f=None, file=None, i=1)

$ python argparse_type.py -f 3.14

Namespace(f=3.14, file=None, i=None)
$ python argparse_type.py --file argparse_type.py

Namespace(f=None, file=<open file 'argparse_type.py', mode 'r' at
0x100d886f0>, i=None)
```

如果类型转换失败, argparse 会生成一个异常。TypeError 和 ValueError 异常会自动截获, 并转换为一个简单的错误消息提供给用户。其他异常, 如下一个例子中输入文件不存在时产生的 IOError, 则必须由调用者来处理。

```
$ python argparse_type.py -i a

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: argument -i: invalid int value: 'a'

$ python argparse_type.py -f 3.14.15

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: argument -f: invalid float value: '3.14.15'

$ python argparse_type.py --file does_not_exist.txt

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: [Errno 2] No such file or directory:
'does_not_exist.txt'
```

要限制输入参数为预定义集合中的一个值，可以使用 `choices` 参数。

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('--mode', choices=('read-only', 'read-write'))

print parser.parse_args()
```

如果 `--mode` 的参数不是允许的某个值，会生成一个错误，处理将停止。

```
$ python argparse_choices.py -h

usage: argparse_choices.py [-h] [--mode {read-only,read-write}]
optional arguments:
  -h, --help            show this help message and exit
  --mode {read-only,read-write}

$ python argparse_choices.py --mode read-only

Namespace(mode='read-only')

$ python argparse_choices.py --mode invalid

usage: argparse_choices.py [-h] [--mode {read-only,read-write}]
argparse_choices.py: error: argument --mode: invalid choice:
'invalid' (choose from 'read-only', 'read-write')
```

文件参数

尽管可以用一个字符串参数实例化 `file` 对象，但是未能包括访问模式参数。`FileType` 提供了一种更为灵活的方式来指定一个参数应当是文件，而且包括模式和缓冲区大小。

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-i', metavar='in-file',
                    type=argparse.FileType('rt'))
parser.add_argument('-o', metavar='out-file',
                    type=argparse.FileType('wt'))

try:
    results = parser.parse_args()
    print 'Input file:', results.i
    print 'Output file:', results.o
except IOError, msg:
    parser.error(str(msg))
```



与参数名关联的值是打开文件句柄。不再使用这个文件时，应用要负责关闭文件。

```
$ python argparse_FileType.py -h

usage: argparse_FileType.py [-h] [-i in-file] [-o out-file]
optional arguments:
  -h, --help      show this help message and exit
  -i in-file
  -o out-file

$ python argparse_FileType.py -i argparse_FileType.py -o tmp_file.txt

Input file: <open file 'argparse_FileType.py', mode 'rt' at
0x100d886f0>
Output file: <open file 'tmp_file.txt', mode 'wt' at 0x100dfa150>

$ python argparse_FileType.py -i no_such_file.txt

usage: argparse_FileType.py [-h] [-i in-file] [-o out-file]
argparse_FileType.py: error: [Errno 2] No such file or directory:
'no_such_file.txt'
```

定制动作

除了前面描述的内置动作之外，还可以提供一个实现了 Action API 的对象来定义定制动作。作为动作 (action) 传入 `add_argument()` 的对象要有一些形参描述所定义的参数 (为 `add_argument()` 指定的所有参数)，并返回一个可调用对象，它的形参包括 `parser` (用来处理参数)、`namespace` (包含解析结果)、`value` (所处理参数的值)，以及触发动作的 `option_string`。

已经提供了类 `Action`，可以把它作为定义新动作的起点。构造函数处理参数定义，所以只需要在子类中覆盖 `__call__()`。

```
import argparse

class CustomAction(argparse.Action):
    def __init__(self,
                  option_strings,
                  dest,
                  nargs=None,
                  const=None,
                  default=None,
                  type=None,
                  choices=None,
                  required=False,
                  help=None,
                  metavar=None):
        argparse.Action.__init__(self,
                                  option_strings=option_strings,
                                  dest=dest,
```




```

        nargs=nargs,
        const=const,
        default=default,
        type=type,
        choices=choices,
        required=required,
        help=help,
        metavar=metavar,
    )

    print 'Initializing CustomAction'
    for name,value in sorted(locals().items()):
        if name == 'self' or value is None:
            continue
        print '  %s = %r' % (name, value)
    print
    return

def __call__(self, parser, namespace, values,
             option_string=None):
    print 'Processing CustomAction for "%s"' % self.dest
    print '  parser = %s' % id(parser)
    print '  values = %r' % values
    print '  option_string = %r' % option_string

    # Do some arbitrary processing of the input values
    if isinstance(values, list):
        values = [ v.upper() for v in values ]
    else:
        values = values.upper()
    # Save the results in the namespace using the destination
    # variable given to our constructor.
    setattr(namespace, self.dest, values)
    print

parser = argparse.ArgumentParser()

parser.add_argument('-a', action=CustomAction)
parser.add_argument('-m', nargs='*', action=CustomAction)
results = parser.parse_args(['-a', 'value',
                             '-m', 'multivalue',
                             'second'])

print results

```

values 的类型取决于 nargs 的值。如果参数允许多个值，values 就是一个列表（即使其中只包含一项）。

option_string 的值也取决于最初的参数规范。对于必要的位置参数，option_string 总是 None。

```
$ python argparse_custom_action.py

Initializing CustomAction
dest = 'a'
option_strings = ['-a']
required = False

Initializing CustomAction
dest = 'm'
nargs = '*'
option_strings = ['-m']
required = False

Initializing CustomAction
dest = 'positional'
option_strings = []
required = True

Processing CustomAction for "a"
parser = 4309267472
values = 'value'
option_string = '-a'

Processing CustomAction for "m"
parser = 4309267472
values = ['multivalue', 'second']
option_string = '-m'

Namespace(a='VALUE', m=['MULTIValue', 'SECOND'])
```

参见:

`argparse` (<http://docs.python.org/library/argparse.html>) 这个模块的标准库文档。

`Original argparse` (<http://pypi.python.org/pypi/argparse>) 标准库之外 `argparse` 版本的 PyPI 页面。这个版本与 Python 的较早版本兼容，可以单独安装。

`ConfigParser` (14.8 节) 读写配置文件。

14.4 readline——GNU Readline 库

作用：为 GNU Readline 库提供一个接口，用于在命令提示窗口与用户交互。

Python 版本：1.4 及以后版本

`readline` 模块可以用于改进交互式命令程序，使之更易于使用。这个模块主要用于提供命令行文本完成特性，即“tab 完成”功能 (tab completion)。

注意：由于 `readline` 与控制台内容交互，如果打印调试消息会很难看出哪些是示例代码所做的，

哪些是 readline 自动完成的工作。下面的例子使用 logging 模块将调试信息写到一个单独的文件。每个示例都会显示日志输出。

注意：默认情况下，readline 所需的 GNU 库并非所有平台都提供。如果你的系统未包括这些库，安装依赖库之后可能需要重新编译 Python 解释器，以启用这个模块。

14.4.1 配置

有两种方法配置底层 readline 库，可以使用一个配置文件，或者利用 `parse_and_bind()` 函数。配置选项包括调用完成特性的按键绑定、编译模式（vi 或 emacs），以及其他一些值。有关的详细信息可以参考 GNU Readline 库的文档。

要启用“tab 完成”，最容易的方法就是利用一个对 `parse_and_bind()` 调用。其他选项可以同时设置。下面这个例子会改变编辑控制，将使用“vi”模式而不是默认的“emacs”。要编辑当前输入行，可以按下 ESc 键，然后使用常规的 vi 导航键，如 j、k、l 和 h。

```
import readline

readline.parse_and_bind('tab: complete')
readline.parse_and_bind('set editing-mode vi')
while True:
    line = raw_input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print 'ENTERED: "%s"' % line
```

这个配置可以作为指令存储在一个文件中，由库利用一个调用来读取。如果 `myreadline.rc` 包含

```
# Turn on tab completion
tab: complete

# Use vi editing mode instead of emacs
set editing-mode vi
```

可以用 `read_init_file()` 读取这个文件。

```
import readline

readline.read_init_file('myreadline.rc')

while True:
    line = raw_input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print 'ENTERED: "%s"' % line
```



14.4.2 完成文本

这个程序有一组内置命令，用户输入指令时将使用 tab 完成功能。

```
import readline
import logging

LOG_FILENAME = '/tmp/completer.log'
logging.basicConfig(filename=LOG_FILENAME,
                    level=logging.DEBUG,
                    )

class SimpleCompleter(object):

    def __init__(self, options):
        self.options = sorted(options)
        return

    def complete(self, text, state):
        response = None
        if state == 0:
            # This is the first time for this text,
            # so build a match list.
            if text:
                self.matches = [s
                                for s in self.options
                                if s and s.startswith(text)]
                logging.debug('%s matches: %s',
                              repr(text), self.matches)
            else:
                self.matches = self.options[:]
                logging.debug('(empty input) matches: %s',
                              self.matches)

            # Return the state'th item from the match list,
            # if we have that many.
            try:
                response = self.matches[state]
            except IndexError:
                response = None
            logging.debug('complete(%s, %s) => %s',
                          repr(text), state, repr(response))
        return response

def input_loop():
    line = ''
    while line != 'stop':
        line = raw_input('Prompt ("stop" to quit): ')
```

```

print 'Dispatch %s' % line

# Register the completer function
OPTIONS = ['start', 'stop', 'list', 'print']
readline.set_completer(SimpleCompleter(OPTIONS).complete)

# Use the tab key for completion
readline.parse_and_bind('tab: complete')

# Prompt the user for text
input_loop()

```

`input_loop()` 函数逐行读取，直至输入值为“stop”。更复杂的程序还可以具体解析输入行，并运行命令。

`SimpleCompleter` 类维护了一个“选项”列表，作为自动完成的候选。实例的 `complete()` 方法要向 `readline` 注册为完成源。参数是一个要完成的文本串 (`text`) 和一个状态值 (`state`)，状态值指示对这个文本调用函数的次数。这个函数会反复调用，每次将状态值递增。如果对这个状态值有一个候选动作，则应返回一个串，如果没有更多的候选，则返回 `None`。这里的 `complete()` 实现会在 `state` 为 0 时查找一组匹配，然后在后续调用时返回所有候选匹配，一次返回一个。

运行时，初始输出为：

```
$ python readline_completer.py
```

```
Prompt ("stop" to quit):
```

按两次 Tab，会打印出一个选项列表。

```
$ python readline_completer.py
```

```
Prompt ("stop" to quit):
```

```
list print start stop
```

```
Prompt ("stop" to quit):
```

日志文件显示出这里分别利用两个不同的状态值序列来调用 `complete()`。

```
$ tail -f /tmp/completer.log
```

```

DEBUG:root:(empty input) matches: ['list', 'print', 'start', 'stop']
DEBUG:root:complete('', 0) => 'list'
DEBUG:root:complete('', 1) => 'print'
DEBUG:root:complete('', 2) => 'start'
DEBUG:root:complete('', 3) => 'stop'
DEBUG:root:complete('', 4) => None
DEBUG:root:(empty input) matches: ['list', 'print', 'start', 'stop']
DEBUG:root:complete('', 0) => 'list'
DEBUG:root:complete('', 1) => 'print'

```

```
DEBUG:root:complete('', 2) => 'start'
DEBUG:root:complete('', 3) => 'stop'
DEBUG:root:complete('', 4) => None
```

第一个序列来自第一次按下 Tab 键。完成算法会查询所有候选，不过并不扩展空的输入行。然后，第二次按下 Tab 时，重新计算候选列表，以便打印给用户。

如果下一个输入为“l”，然后是另一个 Tab，屏幕上会显示以下内容：

```
Prompt ("stop" to quit): list
```

日志会反映 complete() 的不同参数。

```
DEBUG:root:'l' matches: ['list']
DEBUG:root:complete('l', 0) => 'list'
DEBUG:root:complete('l', 1) => None
```

现在按下回车 (RETURN)，会导致 raw_input() 返回这个值，while 循环将继续。

```
Dispatch list
```

```
Prompt ("stop" to quit):
```

对于以“s”开头的命令有两种可能的完成情况。键入“s”，然后按下 TAB，可以找到“start”和“stop”是候选，不过现在只能部分完成屏幕上的文本，即增加一个“t”。

日志文件中可以反映这一点。

```
DEBUG:root:'s' matches: ['start', 'stop']
DEBUG:root:complete('s', 0) => 'start'
DEBUG:root:complete('s', 1) => 'stop'
DEBUG:root:complete('s', 2) => None
```

屏幕会显示以下内容。

```
Prompt ("stop" to quit): st
```

警告：如果一个完成器函数产生一个异常，它会悄无声息地将其忽略，readline 会认为没有匹配的完成选择。

14.4.3 访问完成缓冲区

SimpleCompleter 中的完成算法很简化，因为它只查看传入函数的文本参数，而没有使用 readline 的更多内部状态。还可以使用 readline 函数来管理输入缓冲区的文本。

```
import readline
import logging

LOG_FILENAME = '/tmp/completer.log'
logging.basicConfig(filename=LOG_FILENAME,
                    level=logging.DEBUG,
                    )
```

```

first]
n of input
ompleted)
PDG

```

```

        logging.debug('candidates=%s',
                      self.current_candidates)

    except (KeyError, IndexError), err:
        logging.error('completion error: %s', err)
        self.current_candidates = []

    try:
        response = self.current_candidates[state]
    except IndexError:
        response = None
    logging.debug('complete(%s, %s) => %s',
                  repr(text), state, response)
    return response

def input_loop():
    line = ''
    while line != 'stop':
        line = raw_input('Prompt ("stop" to quit): ')
        print 'Dispatch %s' % line
    # Register our completer function
    readline.set_completer(BufferAwareCompleter(
        {'list': ['files', 'directories'],
         'print': ['byname', 'bysize'],
         'stop': [],
        }).complete)

    # Use the tab key for completion
    readline.parse_and_bind('tab: complete')

    # Prompt the user for text .
    input_loop()

```

这个例子要完成有子选项的命令。complete() 方法需要在输入缓冲区中查看完成的位置，来确定它是第一个词还是后一个词的一部分。如果目标是第一个词，将使用选项字典的键作为候选。如果不是第一个词，则使用第一个词在选项字典中查找候选。

这里有 3 个顶层命令，其中两个有子命令。

- list
 - files
 - directories
- print
 - byname
 - bysize

- stop

按照前面同样的动作序列，两次按下 Tab 键会给出 3 个顶层命令。

```
$ python readline_buffer.py
```

```
Prompt ("stop" to quit):
list print stop
Prompt ("stop" to quit):
```

在日志中可以看到：

```
DEBUG:root:origline=''
DEBUG:root:begin=0
DEBUG:root:end=0
DEBUG:root:being_completed=
DEBUG:root:words=[]
DEBUG:root:complete('', 0) => list
DEBUG:root:complete('', 1) => print
DEBUG:root:complete('', 2) => stop
DEBUG:root:complete('', 3) => None
DEBUG:root:origline=''
DEBUG:root:begin=0
DEBUG:root:end=0
DEBUG:root:being_completed=
DEBUG:root:words=[]
DEBUG:root:complete('', 0) => list
DEBUG:root:complete('', 1) => print
DEBUG:root:complete('', 2) => stop
DEBUG:root:complete('', 3) => None
```

如果第一个词是“list”（单词后面有一个空格），完成候选是不同的。

```
Prompt ("stop" to quit): list
directories files
```

从日志可以看到，所完成的文本不是一整行，只是 list 后的部分。

```
DEBUG:root:origline='list '
DEBUG:root:begin=5
DEBUG:root:end=5
DEBUG:root:being_completed=
DEBUG:root:words=['list']
DEBUG:root:candidates=['files', 'directories']
DEBUG:root:complete('', 0) => files
DEBUG:root:complete('', 1) => directories
DEBUG:root:complete('', 2) => None
DEBUG:root:origline='list '
DEBUG:root:begin=5
DEBUG:root:end=5
DEBUG:root:being_completed=
DEBUG:root:words=['list']
```

```

DEBUG:root:candidates=['files', 'directories']
DEBUG:root:complete('', 0) => files
DEBUG:root:complete('', 1) => directories
DEBUG:root:complete('', 2) => None

```

14.4.4 输入历史

readline 会自动跟踪输入历史。有两组不同的函数来处理历史。当前会话的历史可以用 `get_current_history_length()` 和 `get_history_item()` 访问。这个历史可以保存到一个文件中，以后用 `write_history_file()` 和 `read_history_file()` 重新加载。默认会保存整个历史，不过可以用 `set_history_length()` 设置文件的最大长度。长度为 -1 表示没有限制。

```

import readline
import logging
import os

LOG_FILENAME = '/tmp/completer.log'
HISTORY_FILENAME = '/tmp/completer.hist'

logging.basicConfig(filename=LOG_FILENAME,
                    level=logging.DEBUG,
                    )

def get_history_items():
    num_items = readline.get_current_history_length() + 1
    return [ readline.get_history_item(i)
             for i in xrange(1, num_items)
            ]

class HistoryCompleter(object):

    def __init__(self):
        self.matches = []
        return

    def complete(self, text, state):
        response = None
        if state == 0:
            history_values = get_history_items()
            logging.debug('history: %s', history_values)
            if text:
                self.matches = sorted(h
                                     for h in history_values
                                     if h and h.startswith(text))
            else:
                self.matches = []
            logging.debug('matches: %s', self.matches)

```

```

    try:
        response = self.matches[state]
    except IndexError:
        response = None
    logging.debug('complete(%s, %s) => %s',
                  repr(text), state, repr(response))
    return response

def input_loop():
    if os.path.exists(HISTORY_FILENAME):
        readline.read_history_file(HISTORY_FILENAME)
    print 'Max history file length:', readline.get_history_length()
    print 'Start-up history:', get_history_items()
    try:
        while True:
            line = raw_input('Prompt ("stop" to quit): ')
            if line == 'stop':
                break
            if line:
                print 'Adding "%s" to the history' % line
    finally:
        print 'Final history:', get_history_items()
        readline.write_history_file(HISTORY_FILENAME)

# Register our completer function
readline.set_completer(HistoryCompleter().complete)

# Use the tab key for completion
readline.parse_and_bind('tab: complete')

# Prompt the user for text
input_loop()

```

HistoryCompleter 记住键入的所有内容，并在完成后续输入时使用这些值。

```
$ python readline_history.py
```

```

Max history file length: -1
Start-up history: []
Prompt ("stop" to quit): foo
Adding "foo" to the history
Prompt ("stop" to quit): bar
Adding "bar" to the history
Prompt ("stop" to quit): blah
Adding "blah" to the history
Prompt ("stop" to quit): b
bar  blah
Prompt ("stop" to quit): b

```

```
Prompt ("stop" to quit): stop
Final history: ['foo', 'bar', 'blah', 'stop']
```

按下“b”后再按下两次 Tab 键，日志会显示以下输出。

```
DEBUG:root:history: ['foo', 'bar', 'blah']
DEBUG:root:matches: ['bar', 'blah']
DEBUG:root:complete('b', 0) => 'bar'
DEBUG:root:complete('b', 1) => 'blah'
DEBUG:root:complete('b', 2) => None
DEBUG:root:history: ['foo', 'bar', 'blah']
DEBUG:root:matches: ['bar', 'blah']
DEBUG:root:complete('b', 0) => 'bar'
DEBUG:root:complete('b', 1) => 'blah'
DEBUG:root:complete('b', 2) => None
```

第二次运行脚本时，将从文件读取所有历史。

```
$ python readline_history.py

Max history file length: -1
Start-up history: ['foo', 'bar', 'blah', 'stop']
Prompt ("stop" to quit):
```

还有一些函数可以删除单个历史项，也可以清除整个历史。

14.4.5 hook

模块提供了一些 hook 来触发动作，可以作为交互序列的一部分。打印提示符之前会调用启动 (start-up) hook，显示提示符之后但从用户读取文本之前会运行预输入 (preinput) hook。

```
import readline

def startup_hook():
    readline.insert_text('from start up_hook')

def pre_input_hook():
    readline.insert_text(' from pre_input_hook')
    readline.redisplay()

readline.set_startup_hook(startup_hook)
readline.set_pre_input_hook(pre_input_hook)
readline.parse_and_bind('tab: complete')

while True:
    line = raw_input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print 'ENTERED: "%s"' % line
```

可以充分利用机会，在各个 hook 中使用 `insert_text()` 修改输入缓冲区。

```
$ python readline_hooks.py
```

```
Prompt ("stop" to quit): from startup_hook from pre_input_hook
```

如果在预输入 hook 中修改缓冲区，必须调用 `redisplay()` 更新屏幕。

参见：

`readline` (<http://docs.python.org/library/readline.html>) 这个模块的标准库文档。

GNU `readline` (<http://tiswww.case.edu/php/chet/readline/readline.html>) GNU `Readline` 库的文档。

`readline` init file format (<http://tiswww.case.edu/php/chet/readline/readline.html#SEC10>) 初始化和配置文件格式。

`effbot`: The `readline` module(<http://sandbox.effbot.org/librarybook/readline.htm>) `effbot` 提供的 `readline` 模块指南。

`pyreadline` (<https://launchpad.net/pyreadline>) `pyreadline`，作为基于 Python 的 `readline` 替代库，在 `iPython` 中使用 (<http://ipython.scipy.org/>)。

`cmd` (14.6 节) `cmd` 模块在命令接口中大量使用 `readline` 实现 tab 完成。这里的一些例子就是改自 `cmd` 中的代码。

`rlcompleter` `rlcompleter` 使用 `readline` 为交互式 Python 解释器添加 tab 完成功能。

14.5 getpass——安全密码提示

作用：提示用户输入一个值，通常是一个密码，但不在控制台回显键入的内容。

Python 版本：1.5.2 及以后版本

很多程序通过终端与用户交互，这些程序需要向用户询问密码值，但不在屏幕上显示用户键入的内容。`getpass` 模块提供了一种可移植的方法，可以安全地处理这种口令提示。

14.5.1 示例

`getpass()` 函数会打印一个提示语，然后从用户读取输入，直至用户按下回车。输入会作为一个字符串返回给调用者。

```
import getpass

try:
    p = getpass.getpass()
except Exception, err:
    print 'ERROR:', err
else:
    print 'You entered:', p
```

如果调用者没有指定提示语，默认的提示语为 “Password:”。

```
$ python getpass_defaults.py
```

```
Password:
You entered: sekret
```

可以把这个提示语改为所需的任何值。

```
import getpass

p = getpass.getpass(prompt='What is your favorite color? ')
if p.lower() == 'blue':
    print 'Right. Off you go.'
else:
    print 'Auuuuugh!'
```

有些程序要求输入一个“通行短语”(pass phrase)，而不是一个简单的密码，从而提供更好的安全性。

```
$ python getpass_prompt.py
```

```
What is your favorite color?
Right. Off you go.
```

```
$ python getpass_prompt.py
```

```
What is your favorite color?
Auuuuugh!
```

默认情况下，`getpass()` 使用 `sys.stdout` 来打印提示语字符串。如果程序会在 `sys.stdout` 生成有用的输出，将提示语发送到另一个流通常会更好，如 `sys.stderr`。

```
import getpass
import sys

p = getpass.getpass(stream=sys.stderr)
print 'You entered:', p
```

对提示语使用 `sys.stderr` 表示标准输出可以重定向（到一个管道或一个文件），而不会看到密码提示。用户输入的值仍然不会在屏幕上回显。

```
$ python getpass_stream.py >/dev/null
```

```
Password:
```

14.5.2 无终端使用 `getpass`

在 UNIX 下，`getpass()` 往往需要一个 `tty`，它通过 `termios` 控制这个 `tty`，从而禁用输入回显。这说明，不会从一个重定向到标准输入的非终端流读取值。标准输入重定向时，其结果可能根据 Python 版本而有所不同。如果替换了 `sys.stdin`，Python 2.5 会生成一个异常。

```
$ echo "not sekret" | python2.5 getpass_defaults.py
```

```
ERROR: (25, 'Inappropriate ioctl for device')
```

Python 2.6 和 2.7 得到了改进，会进一步尝试访问 `tty` 来完成处理，如果可以访问则不会产生错误。

```
$ echo "not sekret" | python2.7 getpass_defaults.py
```

```
Password:
You entered: sekret
```

要由调用者检测输入流并非一个 `tty` 的情况，并在这种情况下使用一个候选方法来读取。

```
import getpass
import sys

if sys.stdin.isatty():
    p = getpass.getpass('Using getpass: ')
else:
    print 'Using readline'
    p = sys.stdin.readline().rstrip()
```

```
print 'Read: ', p
```

如果有 `tty`:

```
$ python ./getpass_noterminal.py
```

```
Using getpass:
Read: sekret
```

如果没有 `tty`:

```
$ echo "sekret" | python ./getpass_noterminal.py
```

```
Using readline
Read: sekret
```

参见:

`getpass` (<http://docs.python.org/library/getpass.html>) 这个模块的标准库文档。

`readline` (14.4 节) 交互式提示库。

14.6 cmd——面向行的命令处理器

作用：创建面向行的命令处理器。

Python 版本：1.4 及以后版本

`cmd` 模块包含一个公共类 `Cmd`，这个类要用作交互式 `shell` 和其他命令解释器的基类。默认情况下，它使用 `readline` 完成交互式提示处理、命令行编辑和命令完成。

14.6.1 处理命令

用 `Cmd` 创建的命令解释器使用一个循环从其输入读取所有行，进行解析，然后将命令分派到一个适当的命令处理程序（command handler）。输入行会解析为两部分：命令以及该行上的所有其他文本。如果用户输入 `foo bar`，而且解释器类包含一个名为 `do_foo()` 的方法，则会调用这个方法并以“`bar`”作为其惟一参数。

文件末尾（end-of-file）标志分派至 `do_EOF()`。如果一个命令处理程序返回 `true` 值，程序会妥善地退出。所以要提供一个简洁的方法退出解释器，就一定要实现 `do_EOF()`，并让它返回 `True`。

这个简单的示例程序支持“`greet`”命令。

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    def do_greet(self, line):
        print "hello"

    def do_EOF(self, line):
        return True

if __name__ == '__main__':
    HelloWorld().cmdloop()
```

交互式地运行这个程序，可以展示如何分派命令，并显示 `Cmd` 中包含的一些特性。

```
$ python cmd_simple.py
```

```
(Cmd)
```

首先要注意的是命令提示语（`Cmd`）。这个提示语可以通过属性 `prompt` 来配置。如果由于一个命令处理器而改变了提示语，将使用新值来询问下一个命令。

```
(Cmd) help
```

```
Undocumented commands:
=====
EOF greet help
```

`help` 命令内置在 `Cmd` 中。如果没有提供参数，`help` 会显示可用命令的列表。如果输入包括一个命令名，输出则更为详细，将只显示该命令的详细信息（如果有这个命令）。

如果命令是 `greet`，会调用 `do_greet()` 来进行处理。

```
(Cmd) greet
hello
```


对应一个命令，如果类中没有包含特定的命令处理器，会调用方法 `default()`，并以整个输入行作为参数。`default()` 的内置实现会报告一个错误。

```
(Cmd) foo
*** Unknown syntax: foo
```

由于 `do_EOF()` 返回 `True`，键入 `Ctrl-D` 会使解释器退出。

```
(Cmd) ^D$
```

退出时不会打印换行，所以结果看上去有些乱。

14.6.2 命令参数

这个例子做了一些改进，来消除存在的一些问题，并为 `greet` 命令增加帮助。

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""
    def do_greet(self, person):
        """greet [person]
        Greet the named person"""
        if person:
            print "hi,", person
        else:
            print 'hi'

    def do_EOF(self, line):
        return True

    def postloop(self):
        print

if __name__ == '__main__':
    HelloWorld().cmdloop()
```

增加到 `do_greet()` 的 docstring 会成为这个命令的帮助文本。

```
$ python cmd_arguments.py
```

```
(Cmd) help
```

```
Documented commands (type help ):
```

```
=====
greet
```

```
Undocumented commands:
```

```
=====
EOF help
```



```
(Cmd) help greet
greet [person]
    Greet the named person
```

输出显示了 `greet` 的一个可选参数 `person`。尽管这个参数对命令来说是可选的，但是命令和回调方法之间存在一个区别。方法总是有参数，不过有时这个值是一个空串。要由命令处理器来确定空参数是否合法，或者对命令做进一步的解析和处理。在这个例子中，如果提供了一个人名，就会提供个性化的欢迎词。

```
(Cmd) greet Alice
hi, Alice
(Cmd) greet
hi
```

不论用户是否指定参数，传递到命令处理器的值都不会包含命令本身。这会简化命令处理器中的解析，特别是在需要多个参数时。

14.6.3 现场帮助

在前面的例子中，帮助文本的格式化还需要有所改进。由于它来自 `docstring`，所以保留了源文件中的缩进。可以修改源文件，删除多余的空白符，不过这会使应用代码看起来格式很糟糕。更好的解决方案是为 `greet` 命令实现一个帮助处理程序，名为 `help_greet()`。将调用这个帮助处理程序为指定的命令生成帮助文本。

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    def do_greet(self, person):
        if person:
            print "hi,", person
        else:
            print 'hi'

    def help_greet(self):
        print '\n'.join([ 'greet [person]',
                           'Greet the named person',
                           ])

    def do_EOF(self, line):
        return True

if __name__ == '__main__':
    HelloWorld().cmdloop()
```

在这个例子中，文本是静态的，不过得到了更好的格式化。还可以使用前面的命令状态将

帮助文档的内容调整到当前上下文。

```
$ python cmd_do_help.py

(Cmd) help greet
greet [person]
Greet the named person
```

要由帮助处理程序具体输出帮助消息，而不只是返回帮助文本在别处处理。

14.6.4 自动完成

Cmd 利用处理器方法支持根据命令名实现命令完成。用户在输入提示符处按下 Tab 键就会触发这个完成功能。可能有多个完成候选时，按下两次 Tab 键会显示一个选项列表。

```
$ python cmd_do_help.py

(Cmd) <tab><tab>
EOF      greet  help
(Cmd) h<tab>
(Cmd) help
```

一旦命令已知，可以由带 `complete_` 前缀的方法来处理参数完成。这就允许新的完成处理器使用任意的规则组装一个完成候选列表（也就是说，可以查询数据库，或者查看一个文件或文件系统上的目录）。在这里，程序硬编码编写了一个“朋友”集合，与命名或匿名的陌生人相比，这些朋友会收到更亲切的欢迎。实际程序可能会在某个地方保存这个列表，读取一次后将内容缓存，以便在需要时查看。

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    FRIENDS = [ 'Alice', 'Adam', 'Barbara', 'Bob' ]

    def do_greet(self, person):
        "Greet the person"
        if person and person in self.FRIENDS:
            greeting = 'hi, %s!' % person
        elif person:
            greeting = "hello, " + person
        else:
            greeting = 'hello'
        print greeting

    def complete_greet(self, text, line, begidx, endidx):
        if not text:
            completions = self.FRIENDS[:]
```



```

        else:
            completions = [ f
                            for f in self.FRIENDS
                            if f.startswith(text)
                            ]
            return completions

    def do_EOF(self, line):
        return True

if __name__ == '__main__':
    HelloWorld().cmdloop()

```

如果有输入文本，`complete_greet()` 会返回一个匹配的朋友列表。否则，返回整个朋友列表。

```
$ python cmd_arg_completion.py
```

```

(Cmd) greet <tab><tab>
Adam    Alice    Barbara  Bob
(Cmd) greet A<tab><tab>
Adam    Alice
(Cmd) greet Ad<tab>
(Cmd) greet Adam
hi, Adam!

```

如果给定的名字不在朋友列表中，会给出正式的欢迎词。

```

(Cmd) greet Joe
hello, Joe

```

14.6.5 覆盖基类方法

`Cmd` 包括的很多方法可以覆盖为 hook，用来采取动作或改变基类行为。下面这个例子并不详尽，不过其中包含很多通常很有用的方法。

```

import cmd

class Illustrate(cmd.Cmd):
    "Illustrate the base class method use."

    def cmdloop(self, intro=None):
        print 'cmdloop(%s)' % intro
        return cmd.Cmd.cmdloop(self, intro)

    def preloop(self):
        print 'preloop()'

    def postloop(self):
        print 'postloop()'

```



```

def parseline(self, line):
    print 'parseline(%s) =>' % line,
    ret = cmd.Cmd.parseline(self, line)
    print ret
    return ret

def onecmd(self, s):
    print 'onecmd(%s)' % s
    return cmd.Cmd.onecmd(self, s)

def emptyline(self):
    print 'emptyline()'
    return cmd.Cmd.emptyline(self)

def default(self, line):
    print 'default(%s)' % line
    return cmd.Cmd.default(self, line)

def precmd(self, line):
    print 'precmd(%s)' % line
    return cmd.Cmd.precmd(self, line)
def postcmd(self, stop, line):
    print 'postcmd(%s, %s)' % (stop, line)
    return cmd.Cmd.postcmd(self, stop, line)

def do_greet(self, line):
    print 'hello,', line

def do_EOF(self, line):
    "Exit"
    return True

if __name__ == '__main__':
    Illustrate().cmdloop('Illustrating the methods of cmd.Cmd')

```

cmdloop() 是解释器的主处理循环。通常没有必要覆盖这个循环，因为可以使用 preloop() 和 postloop() hook。

每次 cmdloop() 迭代都会调用 onecmd()，将命令分派到其处理器。实际输入行用 parseline() 解析来创建一个元组，其中包含命令和该行上的其余部分。

如果这一行为空，则调用 emptyline()。默认实现会再次运行前面的命令。如果这一行包含一个命令，将调用第一个 precmd()，然后查看并调用处理器。如果没有找到，则调用 default()。最后调用 postcmd()。

以下是添加了 print 语句的一个示例会话。

```
$ python cmd_illustrate_methods.py
```

```

cmdloop(Illustrating the methods of cmd.Cmd)
preloop()
Illustrating the methods of cmd.Cmd
(Cmd) greet Bob
precmd(greet Bob)
onecmd(greet Bob)
parseline(greet Bob) => ('greet', 'Bob', 'greet Bob')
hello, Bob
postcmd(None, greet Bob)
(Cmd) ^Dprecmd(EOF)
onecmd(EOF)
parseline(EOF) => ('EOF', '', 'EOF')
postcmd(True, EOF)
postloop()

```

14.6.6 通过属性配置 Cmd

除了前面描述的方法，还有很多属性可以用来控制命令解释器。可以把 `prompt` 设置为一个字符串，每次要求用户输入一个新命令时就显示这个字符串。`intro` 是程序开始时打印的“欢迎”消息。`cmdloop()` 取对应这个值的一个参数，或者也可以在类上直接设置。打印帮助时，可以用 `doc_header`、`misc_header`、`undoc_header` 和 `ruler` 属性对输出进行格式化。

```

import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    prompt = 'prompt: '
    intro = "Simple command processor example."

    doc_header = 'doc_header'
    misc_header = 'misc_header'
    undoc_header = 'undoc_header'

    ruler = '-'

    def do_prompt(self, line):
        "Change the interactive prompt"
        self.prompt = line + ': '

    def do_EOF(self, line):
        return True

if __name__ == '__main__':
    HelloWorld().cmdloop()

```

这个示例类显示了一个命令处理器，允许用户控制交互式会话的提示语。



```
$ python cmd_attributes.py

Simple command processor example.
prompt: prompt hello
hello: help
doc_header
-----
prompt

undoc_header
-----
EOF help

hello:
```

14.6.7 运行 shell 命令

作为对标准命令处理的补充，Cmd 包括两个特殊的命令前缀。问号 (?) 等价于内置的 help 命令，可以用同样的方式使用。感叹号 (!) 对应 do_shell()，它要“作为外壳”运行其他命令，如下例所示。

```
import cmd
import subprocess

class ShellEnabled(cmd.Cmd):

    last_output = ''

    def do_shell(self, line):
        "Run a shell command"
        print "running shell command:", line
        sub_cmd = subprocess.Popen(line,
                                    shell=True,
                                    stdout=subprocess.PIPE)
        output = sub_cmd.communicate()[0]
        print output
        self.last_output = output

    def do_echo(self, line):
        """Print the input, replacing '$out' with
        the output of the last shell command.
        """
        # Obviously not robust
        print line.replace('$out', self.last_output)

    def do_EOF(self, line):
        return True
```

```
if __name__ == '__main__':
    ShellEnabled().cmdloop()
```

这个 `echo` 命令实现将其参数中的串 `$out` 替换为前面 `shell` 命令的输出。

```
$ python cmd_do_shell.py
```

```
(Cmd) ?
```

```
Documented commands (type help ):
```

```
=====
echo  shell
```

```
Undocumented commands:
```

```
=====
EOF  help
```

```
(Cmd) ? shell
```

```
Run a shell command
```

```
(Cmd) ? echo
```

```
Print the input, replacing '$out' with
the output of the last shell command
```

```
(Cmd) shell pwd
```

```
running shell command: pwd
```

```
/Users/dhellmann/Documents/PyMOTW/in_progress/cmd
```

```
(Cmd) ! pwd
```

```
running shell command: pwd
```

```
/Users/dhellmann/Documents/PyMOTW/in_progress/cmd
```

```
(Cmd) echo $out
```

```
/Users/dhellmann/Documents/PyMOTW/in_progress/cmd
```

```
(Cmd)
```

14.6.8 候选输入

`Cmd()` 的默认模式是通过 `readline` 库与用户交互，不过也可以使用标准 UNIX `shell` 重定向为标准输入传递一系列命令。

```
$ echo help | python cmd_do_help.py
```

```
(Cmd)
```

```
Documented commands (type help ):
```

```
=====
greet
```

```
Undocumented commands:
```

```
=====
```



```
EOF help
```

```
(Cmd)
```

要让程序直接读取一个脚本文件，可能还需要另外一些修改。因为 `readline` 与终端/tty 设备交互，而不是与标准输入流交互，从文件读取脚本时应当将其禁用。另外，为了避免打印多余的提示语，可以把提示语设置为一个空串。下面这个例子显示了如何打开一个文件，并作为输入将它传递到 `HelloWorld` 例子的一个修改版本。

```
import cmd

class HelloWorld(cmd.Cmd):
    """Simple command processor example."""

    # Disable rawinput module use
    use_rawinput = False

    # Do not show a prompt after each command read
    prompt = ''

    def do_greet(self, line):
        print "hello,", line

    def do_EOF(self, line):
        return True

if __name__ == '__main__':
    import sys
    with open(sys.argv[1], 'rt') as input:
        HelloWorld(stdin=input).cmdloop()
```

将 `use_rawinput` 设置为 `False`，`prompt` 设置为一个空串，现在可以在这个输入文件上调用这个脚本。

```
greet
greet Alice and Bob
```

它会生成以下输出：

```
$ python cmd_file.py cmd_file.txt

hello,
hello, Alice and Bob
```

14.6.9 `sys.argv` 的命令

也可以将程序的命令行参数处理为命令，提供给解释器类，而不是从控制台或文件读取命令。要使用命令行参数，可以直接调用 `onecmd()`，如下例所示。

```
import cmd
```

```

class InteractiveOrCommandLine(cmd.Cmd):
    """Accepts commands via the normal interactive
    prompt or on the command line.
    """

    def do_greet(self, line):
        print 'hello,', line

    def do_EOF(self, line):
        return True

if __name__ == '__main__':
    import sys
    if len(sys.argv) > 1:
        InteractiveOrCommandLine().onecmd(' '.join(sys.argv[1:]))
    else:
        InteractiveOrCommandLine().cmdloop()

```

由于 `onecmd()` 取一个字符串作为输入，在参数传入之前，需要把程序的参数连接起来。

```
$ python cmd_argv.py greet Command-Line User
```

```
hello, Command-Line User
$ python cmd_argv.py
```

```
(Cmd) greet Interactive User
hello, Interactive User
(Cmd)
```

参见：

`cmd` (<http://docs.python.org/library/cmd.html>) 这个模块的标准库文档。

`cmd2` (<http://pypi.python.org/pypi/cmd2>) 直接替换 `cmd`，提供了额外的特性。

`GNU Readline` (<http://tiswww.case.edu/php/chet/readline/rltop.html>) `GNU Readline` 库提供了一些函数，允许用户在键入时编辑输入行。

`readline` (14.4 节) `readline` 的 Python 标准库接口。

`subprocess` (10.1 节) 管理其他进程及其输出。

14.7 shlex——解析 shell 语法

作用：shell 语法的词法分析。

Python 版本：1.5.2 及以后版本

`shlex` 模块实现了一个类来解析简单的类 shell 语法，可以用来编写领域特定的语言，或者解析加引号的字符串（这个任务没有表面看起来那么简单）。

14.7.1 加引号的字符串

处理输入文本时有一个常见的问题，往往要把一个加引号的单词序列标识为一个实体。根据引号划分文本可能与预想的并不一样，特别是嵌套有多层引号时。以下面的文本为例。

```
This string has embedded "double quotes" and 'single quotes' in it,
and even "a 'nested example'".
```

一种简单的方法是构造一个正则表达式，来查找引号之外的文本部分，将它们与引号内的文本分开，或者反之。这可能带来不必要的复杂性，而且很容易因为边界条件出错，如撇号或者拼写错误。更好的解决方案是使用一个真正的解析器，如 shlex 模块提供的解析器。以下是一个简单的例子，它使用 shlex 类打印输入文件中找到的 token。

```
import shlex
import sys

if len(sys.argv) != 2:
    print 'Please specify one filename on the command line.'
    sys.exit(1)

filename = sys.argv[1]
body = file(filename, 'rt').read()
print 'ORIGINAL:', repr(body)
print

print 'TOKENS:'
lexer = shlex.shlex(body)
for token in lexer:
    print repr(token)
```

对包含嵌入引号的数据运行这个解析器时，会得到期望的 token 列表。

```
$ python shlex_example.py quotes.txt
```

```
ORIGINAL: 'This string has embedded "double quotes" and \'single quotes\'
in it,\nand even "a \'nested example\'".\n'
```

```
TOKENS:
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
','
'and'
```

```
'even'
'a \'nested example\''
','
```

孤立的引号（如撇号）也会处理。考虑以下输入文件。

```
This string has an embedded apostrophe, doesn't it?
```

完全可以找出包含嵌入撇号的 token。

```
$ python shlex_example.py apostrophe.txt
```

```
ORIGINAL: "This string has an embedded apostrophe, doesn't it?"
```

```
TOKENS:
'This'
'string'
'has'
'an'
'embedded'
'apostrophe'
','
'doesn't'
'it'
'?'
```

14.7.2 嵌入注释

由于解析器要用于处理命令语言，所以也需要处理注释。默认情况下，# 后面的文本会认为是注释的一部分，并被忽略。由于解析器的特点，它只支持单字符注释前缀。可以通过 `commenters` 属性配置使用的注释字符集。

```
$ python shlex_example.py comments.txt
```

```
ORIGINAL: 'This line is recognized.\n# But this line is ignored.\nAnd this line is processed.'
```

```
TOKENS:
'This'
'line'
'is'
'recognized'
','
'And'
'this'
'line'
'is'
'processed'
','
```



14.7.3 分解

要把一个现有的字符串分解为其组成 token，可以使用便利函数 `split()`，这是解析器的一个简单包装器。

```
import shlex

text = """This text has "quoted parts" inside it."""
print 'ORIGINAL:', repr(text)
print

print 'TOKENS:'
print shlex.split(text)

结果是一个列表。
$ python shlex_split.py

ORIGINAL: 'This text has "quoted parts" inside it.'

TOKENS:
['This', 'text', 'has', 'quoted parts', 'inside', 'it.']
```

14.7.4 包含其他 Token 源

`shlex` 类包括很多配置属性来控制其行为。`source` 属性可以启用代码（或配置）重用特性，允许一个 token 流包含另一个 token 流。这类似于 Bourne shell 的 `source` 操作符，也因此得名。

```
import shlex

text = """This text says to source quotes.txt before continuing."""
print 'ORIGINAL:', repr(text)
print

lexer = shlex.shlex(text)
lexer.wordchars += '.'
lexer.source = 'source'

print 'TOKENS:'
for token in lexer:
    print repr(token)
```

原文本中的字符串 `source quotes.txt` 会得到特殊处理。由于 `lexer` 的 `source` 属性设置为“`source`”，遇到这个关键字时，会自动包含下一行上出现的文件名。为了让文件名作为单个 token 出现，需要在单词所包含字符的列表中添加 `.` 字符（否则“`quotes.txt`”会变成 3 个 token：“`quotes`”、“`.`”和“`txt`”）。输出如下：

```
$ python shlex_source.py
```

```
ORIGINAL: 'This text says to source quotes.txt before continuing.'
```

```
TOKENS:
'This'
'text'
'says'
'to'
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
','
'and'
'even'
'"a \'nested example\'"'
'.'
'before'
'continuing.'
```

“source”特性使用了一个名为 `sourcehook()` 的方法加载额外的输入源，所以 `shlex` 的子类可以提供一个候选实现，从非文件的其他位置加载数据。

14.7.5 控制解析器

前面的例子展示了可以改变 `wordchars` 值来控制单词中包含哪些字符。还可以设置 `quotes` 字符来使用额外或替代引号。每个引号必须是单个字符，所以不可能有不同的开始和结束引号（例如，不会解析括号）。

```
import shlex

text = """|Col 1||Col 2||Col 3|"""
print 'ORIGINAL:', repr(text)
print

lexer = shlex.shlex(text)
lexer.quotes = '|'

print 'TOKENS:'
for token in lexer:
    print repr(token)
```

在这个例子中，每个表单元格用竖线包围。



```
$ python shlex_table.py

ORIGINAL: '|Col 1||Col 2||Col 3|'

TOKENS:
'|Col 1|'
'|Col 2|'
'|Col 3|'
```

还可以控制用来分解单词的空白字符。

```
import shlex
import sys

if len(sys.argv) != 2:
    print 'Please specify one filename on the command line.'
    sys.exit(1)

filename = sys.argv[1]
body = file(filename, 'rt').read()
print 'ORIGINAL:', repr(body)
print

print 'TOKENS:'
lexer = shlex.shlex(body)
lexer.whitespace += '.,'
for token in lexer:
    print repr(token)
```

如果 shlex_example.py 中的例子修改为包含点号和逗号，结果会改变。

```
$ python shlex_whitespace.py quotes.txt
```

```
ORIGINAL: 'This string has embedded "double quotes" and \'single quotes\'
in it,\nand even "a \'nested example\''.\n'
```

```
TOKENS:
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
'"single quotes"'
'in'
'it'
'and'
'even'
'"a \'nested example\''"
```



14.7.6 错误处理

所有加引号的串结束之前，如果解析器提前遇到输入末尾，则会产生 `ValueError`。出现这种情况时，可以检查解析器处理输入时维护的一些属性，这很有用。例如，`infile` 指示所处理的文件的名称（如果一个文件用 `source` 包含另一个文件，则可能与原文件不同）。`lineno` 会报告发现错误时的文本行。`lineno` 通常是文件末尾，这可能与第一个引号相距很远。`token` 属性包含尚未包括在一个合法 `token` 中的文本缓冲区。`error_leader()` 方法会用类似 UNIX 编译器的样式生成一个消息前缀，这将启用编辑器（如 `emacs`）来解析错误，并向用户指示有问题的那一行。

```
import shlex

text = """This line is ok.
This line has an "unfinished quote.
This line is ok, too.
"""

print 'ORIGINAL:', repr(text)
print

lexer = shlex.shlex(text)

print 'TOKENS:'
try:
    for token in lexer:
        print repr(token)
except ValueError, err:
    first_line_of_error = lexer.token.splitlines()[0]
    print 'ERROR:', lexer.error_leader(), str(err)
    print 'following "' + first_line_of_error + '"'
```

这个例子会生成以下输出：

```
$ python shlex_errors.py
```

```
ORIGINAL: 'This line is ok.\nThis line has an "unfinished quote.\nThis line is ok, too.\n'
```

```
TOKENS:
```

```
'This'
'line'
'is'
'ok'
'.'
'This'
'line'
'has'
'an'
```




```
ERROR: "None", line 4: No closing quotation
following "unfinished quote."
```

14.7.7 POSIX 与非 POSIX 解析

解析器的默认行为是使用一种向后兼容方式，这不符合 POSIX 规范。要想符合 POSIX 规范，构造解析器时要设置 `posix` 参数。

```
import shlex

for s in [ 'Do"Not"Separate',
           '"Do"Separate',
           'Escaped \e Character not in quotes',
           'Escaped "\e" Character in double quotes',
           "Escaped '\e' Character in single quotes",
           r"Escaped '\'' \"\'\" single quote",
           r"Escaped '\"\" \"\'\" double quote",
           "\"'Strip extra layer of quotes'\"",
           ]:
    print 'ORIGINAL :', repr(s)
    print 'non-POSIX:',

    non_posix_lexer = shlex.shlex(s, posix=False)
    try:
        print repr(list(non_posix_lexer))
    except ValueError, err:
        print 'error(%s)' % err

    print 'POSIX      :',
    posix_lexer = shlex.shlex(s, posix=True)
    try:
        print repr(list(posix_lexer))
    except ValueError, err:
        print 'error(%s)' % err

    print
```

下面几个例子将展示解析行为的差别。

```
$ python shlex_posix.py
```

```
ORIGINAL : 'Do"Not"Separate'
non-POSIX: ['Do"Not"Separate']
POSIX     : ['DoNotSeparate']

ORIGINAL : '"Do"Separate'
non-POSIX: ['"Do"', 'Separate']
```



```

POSIX      : ['DoSeparate']

ORIGINAL : 'Escaped \\e Character not in quotes'
non-POSIX: ['Escaped', '\\', 'e', 'Character', 'not', 'in',
'quotes']
POSIX      : ['Escaped', 'e', 'Character', 'not', 'in', 'quotes']

ORIGINAL : 'Escaped "\\e" Character in double quotes'
non-POSIX: ['Escaped', '"\\e"', 'Character', 'in', 'double',
'quotes']
POSIX      : ['Escaped', '\\e', 'Character', 'in', 'double', 'quotes']

ORIGINAL : "Escaped '\\e' Character in single quotes"
non-POSIX: ['Escaped', "'\\e'", 'Character', 'in', 'single',
'quotes']
POSIX      : ['Escaped', '\\e', 'Character', 'in', 'single', 'quotes']

ORIGINAL : 'Escaped \'\\\'\'\' \'\\\'\'\'\'\' single quote'
non-POSIX: error(No closing quotation)
POSIX      : ['Escaped', '\\ \'\'\'\'\'', 'single', 'quote']

ORIGINAL : 'Escaped "\\\" \"\\\'\'\'\'\' double quote'
non-POSIX: error(No closing quotation)
POSIX      : ['Escaped', '\"', '\\\'\'\'\'', 'double', 'quote']

ORIGINAL : '\"\'Strip extra layer of quotes\'\"'
non-POSIX: ['\"\'Strip extra layer of quotes\'\"']
POSIX      : ['\"Strip extra layer of quotes\'\"]

```

参见:

shlex (<http://docs.python.org/lib/module-shlex.html>) 这个模块的标准库文档。

cmd (14.6 节) 构建交互式命令解释器的工具。

optparse (14.2 节) 命令行选项解析。

getopt (14.1 节) 命令行选项解析。

argparse (14.3 节) 命令行选项解析。

subprocess (10.1 节) 解析命令行后运行命令。

14.8 ConfigParser——处理配置文件

作用：读写类似于 Windows INI 文件的配置文件。

Python 版本：1.5

使用 ConfigParser 模块可以为应用管理用户可编辑的配置文件。配置文件的内容可以组织为组，还支持多个选项值 (option-value) 类型，包括整数、浮点值和布尔值。可以使用 Python

格式化字符串组合选项值来建立更长的值，如由主机名和端口号等较短的值构造 URL。

14.8.1 配置文件格式

ConfigParser 使用的文件格式类似于 Microsoft Windows 较早版本使用的格式。它由一个或多个命名的节 (section) 组成，每一节包含由名和值构成的选项 (option)。

可以通过查找以 “[” 开头并以 “]” 结束的行来标识配置文件的节。中括号之间的值是节名，其中可以包含除中括号以外的任何字符。

在一节中，每行列出一个选项。行以选项名开头，选项名与值之间用一个冒号 (:) 或一个等号 (=) 分开。解析文件时，分隔符两边的空白符会被忽略。

这个示例配置文件有一个名为 “bug_tracker” 的节，其中包含 3 个选项。

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = SECRET
```

14.8.2 读取配置文件

配置文件最常见的用法是允许用户或系统管理员用一个常规的文本编辑器编辑文件，以设置默认的应用行为，然后让应用读取这个文件，并进行解析，根据其内容采取动作。可以使用 SafeConfigParser 的 read() 方法来读取配置文件。

```
from ConfigParser import SafeConfigParser
```

```
parser = SafeConfigParser()
parser.read('simple.ini')
```

```
print parser.get('bug_tracker', 'url')
```

这个程序读取上一节的 simple.ini 文件，打印 bug_tracker 节中 url 选项的值。

```
$ python ConfigParser_read.py
```

```
http://localhost:8080/bugs/
```

read() 方法还接受一个文件名列表。会依次检查各个名，如果文件存在，则打开该文件并读取。

```
from ConfigParser import SafeConfigParser
import glob
```

```
parser = SafeConfigParser()
```

```
candidates = ['does_not_exist.ini', 'also-does-not-exist.ini',
              'simple.ini', 'multisection.ini',
              ]
```

```

found = parser.read(candidates)

missing = set(candidates) - set(found)

print 'Found config files:', sorted(found)
print 'Missing files      :', sorted(missing)

```

`read()` 返回一个列表，其中包含成功加载的文件的名称，所以程序可以发现缺少哪些配置文件，并确定是否将其忽略。

```
$ python ConfigParser_read_many.py
```

```

Found config files: ['multisection.ini', 'simple.ini']
Missing files      : ['also-does-not-exist.ini', 'does_not_exist.ini']

```

Unicode 配置数据

包含 Unicode 数据的配置文件应当使用 `codecs` 模块打开，来设置适当的编码值。将原输入的密码值改为包含 Unicode 字符，并采用 UTF-8 编码保存结果，可以得到：

```

[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = Béc@ét

```

可以将 `codecs` 文件句柄传递到 `readfp()`，后者使用其参数的 `readline()` 方法从文件得到行，并进行解析。

```

from ConfigParser import SafeConfigParser
import codecs

parser = SafeConfigParser()

# Open the file with the correct encoding
with codecs.open('unicode.ini', 'r', encoding='utf-8') as f:
    parser.readfp(f)

password = parser.get('bug_tracker', 'password')

print 'Password:', password.encode('utf-8')
print 'Type      :', type(password)
print 'repr()    :', repr(password)

```

`get()` 返回的值是一个 `unicode` 对象，所以为了安全地打印这个对象，必须将它重新编码为 UTF-8。

```
$ python ConfigParser_unicode.py
```

```

Password: Béc@ét
Type      : <type 'unicode'>
repr()    : u'\xdf\xe9\xe7\xae\xe9\u2020'

```

14.8.3 访问配置设置

SafeConfigParser 包含一些方法来检查所解析配置的结构，包括列出节和选项，以及得到它们的值。下面这个配置文件包含两个节，分别对应不同的 Web 服务。

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = SECRET

[wiki]
url = http://localhost:8080/wiki/
username = dhellmann
password = SECRET
```

这个示例程序使用了一些方法来查看配置数据，包括 sections()、options() 和 items()。

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('multisection.ini')

for section_name in parser.sections():
    print 'Section:', section_name
    print ' Options:', parser.options(section_name)
    for name, value in parser.items(section_name):
        print ' %s = %s' % (name, value)
    print
```

sections() 和 options() 会返回字符串列表，而 items() 返回一个包含名 - 值对的元组列表。

```
$ python ConfigParser_structure.py
```

```
Section: bug_tracker
Options: ['url', 'username', 'password']
url = http://localhost:8080/bugs/
username = dhellmann
password = SECRET
```

```
Section: wiki
Options: ['url', 'username', 'password']
url = http://localhost:8080/wiki/
username = dhellmann
password = SECRET
```

测试值是否存在

要测试一节是否存在，可以使用 has_section()，并传入节名。

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
```



```

parser.read('multisection.ini')
for candidate in [ 'wiki', 'bug_tracker', 'dvcs' ]:
    print '%-12s: %s' % (candidate, parser.has_section(candidate))

```

调用 `get()` 之前先测试一个节是否存在，这样可以避免因缺少数据而导致异常。

```
$ python ConfigParser_has_section.py
```

```

wiki           : True
bug_tracker    : True
dvcs           : False

```

使用 `has_option()` 可以测试一个节中某个选项是否存在。

```

from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('multisection.ini')

SECTIONS = [ 'wiki', 'none' ]
OPTIONS = [ 'username', 'password', 'url', 'description' ]

for section in SECTIONS:
    has_section = parser.has_section(section)
    print '%s section exists: %s' % (section, has_section)
    for candidate in OPTIONS:
        has_option = parser.has_option(section, candidate)
        print '%s.%-12s : %s' % (section,
                                candidate,
                                has_option,
                                )
    print

```

如果这个节不存在，`has_option()` 会返回 `False`。

```
$ python ConfigParser_has_option.py
```

```

wiki section exists: True
wiki.username       : True
wiki.password       : True
wiki.url            : True
wiki.description    : False
none section exists: False
none.username       : False
none.password       : False
none.url            : False
none.description    : False

```

值类型

所有节和选项名都处理为字符串，不过选项值可以是字符串、整数、浮点数或者布尔值。



有一些可能的布尔值可以转换为 true 或 false。下面的示例文件分别包含了各类选项值。

```
[ints]
positive = 1
negative = -5

[floats]
positive = 0.2
negative = -3.14

[booleans]
number_true = 1
number_false = 0
yn_true = yes
yn_false = no
tf_true = true
tf_false = false
onoff_true = on
onoff_false = false
```

SafeConfigParser 不会尝试去了解选项类型，而希望应用使用正确的方法来获取所需类型的值。get() 总会返回一个字符串。使用 getint() 可以得到整数，getfloat() 得到浮点数，使用 getboolean() 得到布尔值。

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('types.ini')

print 'Integers:'
for name in parser.options('ints'):
    string_value = parser.get('ints', name)
    value = parser.getint('ints', name)
    print ' %-12s : %-7r -> %d' % (name, string_value, value)
print '\nFloats:'
for name in parser.options('floats'):
    string_value = parser.get('floats', name)
    value = parser.getfloat('floats', name)
    print ' %-12s : %-7r -> %0.2f' % (name, string_value, value)

print '\nBooleans:'
for name in parser.options('booleans'):
    string_value = parser.get('booleans', name)
    value = parser.getboolean('booleans', name)
    print ' %-12s : %-7r -> %s' % (name, string_value, value)
```

使用示例输入运行这个程序，可以生成以下结果：

```
$ python ConfigParser_value_types.py
```

```

Integers:
    positive      : '1'      -> 1
    negative      : '-5'     -> -5

Floats:
    positive      : '0.2'    -> 0.20
    negative      : '-3.14'  -> -3.14

Booleans:
    number_true   : '1'      -> True
    number_false  : '0'      -> False
    yn_true       : 'yes'    -> True
    yn_false      : 'no'     -> False
    tf_true       : 'true'   -> True
    tf_false      : 'false'  -> False
    onoff_true    : 'on'     -> True
    onoff_false   : 'false'  -> False

```

选项作为标志

通常，解析器要求每个选项都有一个明确的值，不过，如果 `SafeConfigParser` 参数 `allow_no_value` 设置为 `True`，选项可以在输入文件一行上单独出现，用作一个标志。

```

import ConfigParser

# Require values
try:
    parser = ConfigParser.SafeConfigParser()
    parser.read('allow_no_value.ini')
except ConfigParser.ParsingError, err:
    print 'Could not parse:', err

# Allow stand-alone option names
print '\nTrying again with allow_no_value=True'
parser = ConfigParser.SafeConfigParser(allow_no_value=True)
parser.read('allow_no_value.ini')
for flag in [ 'turn_feature_on', 'turn_other_feature_on' ]:
    print
    print flag
    exists = parser.has_option('flags', flag)
    print '  has_option:', exists
    if exists:
        print '      get:', parser.get('flags', flag)

```

选项没有明确的值时，`has_option()` 会报告这个选项存在，`get()` 返回 `None`。

```
$ python ConfigParser_allow_no_value.py
```

```

Could not parse: File contains parsing errors: allow_no_value.ini
[line 2]: 'turn_feature_on\n'

```



```
Trying again with allow_no_value=True
```

```
turn_feature_on
  has_option: True
    get: None
```

```
turn_other_feature_on
  has_option: False
```

14.8.4 修改设置

SafeConfigParser 主要通过从文件读取设置来进行配置, 不过也可以以后填充设置, 调用 `add_section()` 创建一个新的节, 另外可以调用 `set()` 增加或修改一个选项。

```
import ConfigParser

parser = ConfigParser.SafeConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://localhost:8080/bugs')
parser.set('bug_tracker', 'username', 'dhellmann')
parser.set('bug_tracker', 'password', 'secret')

for section in parser.sections():
    print section
    for name, value in parser.items(section):
        print ' %s = %r' % (name, value)
```

所有选项都必须设置为字符串, 即使作为整数、浮点数或布尔值来获取。

```
$ python ConfigParser_populate.py
```

```
bug_tracker
  url = 'http://localhost:8080/bugs'
  username = 'dhellmann'
  password = 'secret'
```

可以用 `remove_section()` 和 `remove_option()` 从 SafeConfigParser 删除节和选项。

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('multisection.ini')

print 'Read values:\n'
for section in parser.sections():
    print section
    for name, value in parser.items(section):
        print ' %s = %r' % (name, value)
```

```
parser.remove_option('bug_tracker', 'password')
parser.remove_section('wiki')
```

```
print '\nModified values:\n'
for section in parser.sections():
    print section
    for name, value in parser.items(section):
        print '  %s = %r' % (name, value)
```

删除一节也会删除其中包含的所有选项。

```
$ python ConfigParser_remove.py
```

Read values:

```
bug_tracker
  url = 'http://localhost:8080/bugs/'
  username = 'dhellmann'
  password = 'SECRET'
wiki
  url = 'http://localhost:8080/wiki/'
  username = 'dhellmann'
  password = 'SECRET'
```

Modified values:

```
bug_tracker
  url = 'http://localhost:8080/bugs/'
  username = 'dhellmann'
```

14.8.5 保存配置文件

用所需的数据填充 `SafeConfigParser` 后，就可以调用 `write()` 方法将它保存到一个文件。这样一来，可以提供一个用户界面来编辑配置设置，而不用编写任何代码管理文件。

```
import ConfigParser
import sys

parser = ConfigParser.SafeConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://localhost:8080/bugs')
parser.set('bug_tracker', 'username', 'dhellmann')
parser.set('bug_tracker', 'password', 'secret')

parser.write(sys.stdout)
```

`write()` 方法取一个类文件的对象作为参数。它采用 INI 格式写出数据，以便再由 `SafeConfigParser` 解析。

```
$ python ConfigParser_write.py

[bug_tracker]
url = http://localhost:8080/bugs
username = dhellmann
password = secret
```

警告： 读取、修改和重写配置文件时，原配置文件中的注释不会保留。

14.8.6 选项搜索路径

SafeConfigParser 查找选项时使用了一个多步搜索过程。

开始搜索选项之前，首先会测试节名。如果这个节不存在，而且名不是特殊值 DEFAULT，则产生一个 NoSectionError 异常。

1. 如果选项名出现在传递到 get() 的 vars 字典中，会返回 vars 的值。
2. 如果选项名出现在指定的节中，则返回该节中的值。
3. 如果选项名出现在 DEFAULT 节中，会返回相应的值。
4. 如果选项名出现在传递到构造函数的 defaults 字典中，会返回相应的值。如果这个名未出现在以上任何位置，则产生 NoOptionError。

可以使用以下配置文件来展示搜索路径行为。

```
[DEFAULT]
file-only = value from DEFAULT section
init-and-file = value from DEFAULT section
from-section = value from DEFAULT section
from-vars = value from DEFAULT section

[sect]
section-only = value from section in file
from-section = value from section in file
from-vars = value from section in file
```

这个测试程序包括配置文件中未指定的一些选项默认设置，并覆盖了文件中定义的一些值。

```
import ConfigParser

# Define the names of the options
option_names = [
    'from-default',
    'from-section', 'section-only',
    'file-only', 'init-only', 'init-and-file',
    'from-vars',
]

# Initialize the parser with some defaults
```

```

parser = ConfigParser.SafeConfigParser(
    defaults={'from-default': 'value from defaults passed to init',
              'init-only': 'value from defaults passed to init',
              'init-and-file': 'value from defaults passed to init',
              'from-section': 'value from defaults passed to init',
              'from-vars': 'value from defaults passed to init',
              })

print 'Defaults before loading file:'
defaults = parser.defaults()
for name in option_names:
    if name in defaults:
        print ' %-15s = %r' % (name, defaults[name])

# Load the configuration file
parser.read('with-defaults.ini')

print '\nDefaults after loading file:'
defaults = parser.defaults()
for name in option_names:
    if name in defaults:
        print ' %-15s = %r' % (name, defaults[name])

# Define some local overrides
vars = {'from-vars': 'value from vars'}
# Show the values of all the options
print '\nOption lookup:'
for name in option_names:
    value = parser.get('sect', name, vars=vars)
    print ' %-15s = %r' % (name, value)

# Show error messages for options that do not exist
print '\nError cases:'
try:
    print 'No such option :', parser.get('sect', 'no-option')
except ConfigParser.NoOptionError, err:
    print str(err)

try:
    print 'No such section:', parser.get('no-sect', 'no-option')
except ConfigParser.NoSectionError, err:
    print str(err)

```

输出显示了各个选项值的来源，并展示了不同来源的默认值如何覆盖现有值。

```
$ python ConfigParser_defaults.py
```

Defaults before loading file:

```

from-default      = 'value from defaults passed to init'
from-section      = 'value from defaults passed to init'
init-only         = 'value from defaults passed to init'
init-and-file     = 'value from defaults passed to init'
from-vars         = 'value from defaults passed to init'

```

Defaults after loading file:

```

from-default      = 'value from defaults passed to init'
from-section      = 'value from DEFAULT section'
file-only         = 'value from DEFAULT section'
init-only         = 'value from defaults passed to init'
init-and-file     = 'value from DEFAULT section'
from-vars         = 'value from DEFAULT section'

```

Option lookup:

```

from-default      = 'value from defaults passed to init'
from-section      = 'value from section in file'
section-only      = 'value from section in file'
file-only         = 'value from DEFAULT section'
init-only         = 'value from defaults passed to init'
init-and-file     = 'value from DEFAULT section'
from-vars         = 'value from vars'

```

Error cases:

```

No such option : No option 'no-option' in section: 'sect'
No such section: No section: 'no-sect'

```

14.8.7 用接合合并值

SafeConfigParser 提供了一个特性，称为拼接（interpolation），可以用来将值接合在一起。如果值包含标准 Python 格式串，用 `get()` 获取这个值时就会触发拼接特性。获取的值中指定的选项会按顺序依次替换为相应的值，直到没有必要再做更多替换。

可以重写本节前面的 URL 例子，改为使用拼接，从而能更容易地只改变部分值。例如，这个配置文件将 URL 的协议、主机名和端口分开，作为单独的选项。

```

[bug_tracker]
protocol = http
server = localhost
port = 8080
url = %(protocol)s://%(server)s:%(port)s/bugs/
username = dhellmann
password = SECRET

```

每次调用 `get()` 时会默认地完成拼接。在 `raw` 参数中传入一个 `true` 值可以获取原值，而不会拼接。

```

from ConfigParser import SafeConfigParser

```

```

parser = SafeConfigParser()
parser.read('interpolation.ini')

```

```

print 'Original value      :', parser.get('bug_tracker', 'url')

parser.set('bug_tracker', 'port', '9090')
print 'Altered port value  :', parser.get('bug_tracker', 'url')

print 'Without interpolation:', parser.get('bug_tracker', 'url',
                                           raw=True)

```

由于值由 `get()` 计算, 改变 `url` 值所用的某个设置也会改变返回值。

```
$ python ConfigParser_interpolation.py
```

```

Original value      : http://localhost:8080/bugs/
Altered port value  : http://localhost:9090/bugs/
Without interpolation: %(protocol)s://%(server)s:%(port)s/bugs/

```

使用默认值

并不要求拼接的值出现在原选项所在的同一节中。默认值可以与覆盖值混合使用。

```

[DEFAULT]
url = %(protocol)s://%(server)s:%(port)s/bugs/
protocol = http
server = bugs.example.com
port = 80

[bug_tracker]
server = localhost
port = 8080
username = dhellmann
password = SECRET

```

采用这个配置, `url` 的值来自 `DEFAULT` 节, 替换从查看 `bug_tracker` 开始, 未找到的值则在 `DEFAULT` 中查找。

```
from ConfigParser import SafeConfigParser
```

```

parser = SafeConfigParser()
parser.read('interpolation_defaults.ini')

```

```
print 'URL:', parser.get('bug_tracker', 'url')
```

`hostname` 和 `port` 值来自 `bug_tracker` 节, 但是 `protocol` 来自 `DEFAULT`。

```
$ python ConfigParser_interpolation_defaults.py
```

```
URL: http://localhost:8080/bugs/
```

替换错误

`MAX_INTERPOLATION_DEPTH` 步骤之后替换停止, 以避免递归引用导致的问题。

```

import ConfigParser

parser = ConfigParser.SafeConfigParser()

parser.add_section('sect')
parser.set('sect', 'opt', '%(opt)s')

try:
    print parser.get('sect', 'opt')
except ConfigParser.InterpolationDepthError, err:
    print 'ERROR:', err

```

如果有太多替换步骤，就会产生一个 `InterpolationDepthError` 异常。

```
$ python ConfigParser_interpolation_recursion.py
```

```

ERROR: Value interpolation too deeply recursive:
      section: [sect]
      option  : opt
      rawval  : %(opt)s

```

缺少值会导致一个 `InterpolationMissingOptionError` 异常。

```

import ConfigParser

parser = ConfigParser.SafeConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://%(server)s:%(port)s/bugs')

try:
    print parser.get('bug_tracker', 'url')
except ConfigParser.InterpolationMissingOptionError, err:
    print 'ERROR:', err

```

由于没有定义 `server` 值，所以无法构造 `url`。

```
$ python ConfigParser_interpolation_error.py
```

```

ERROR: Bad value substitution:
      section: [bug_tracker]
      option  : url
      key     : server
      rawval  : :%(port)s/bugs

```

参见：

`ConfigParser` (<http://docs.python.org/library/configparser.html>) 这个模块的标准库文档。
`codecs` (6.7 节) `codecs` 模块用于读写 Unicode 文件。

14.9 日志——报告状态、错误和信息消息

作用：报告状态、错误和信息消息。

Python 版本：2.3 及以后版本

logging 模块定义了一个标准 API，用来报告应用和库的错误及状态信息。由一个标准库模块提供日志 API 的主要好处在于：所有 Python 模块都可以参与日志记录，所以一个应用的日志还可以包含来自第三方模块的消息。

14.9.1 应用与库中的日志记录

应用开发人员和库作者都可以使用 logging，不过分别有不同的考虑。

应用开发人员要配置 logging 模块，将消息定向到适当的输出通道。可以采用不同的详细程度记录消息，或者记录到不同的目标。可以将日志消息写入文件、HTTP GET/POST 位置、通过 SMTP 写入 email 邮件、写入通用套接字或采用操作系统特定的日志机制，logging 模块中提供了所有这些处理器。对于内置类未能处理的特殊需求，还可以创建定制的日志目标类。

库开发人员也可以使用 logging，而且要做的工作更少。只需要为各个上下文分别创建一个日志记录器实例，使用一个适当的名字，然后使用标准级别记录日志。只要库使用 logging API，并提供一致命名和级别选择，就可以根据需要配置应用来显示或隐藏库的消息。

14.9.2 记入文件

大多数应用都配置为将日志记入文件。使用 basicConfig() 函数建立默认处理器，从而将调试消息写至一个文件。

```
import logging

LOG_FILENAME = 'logging_example.out'
logging.basicConfig(filename=LOG_FILENAME,
                    level=logging.DEBUG,
                    )

logging.debug('This message should go to the log file')

with open(LOG_FILENAME, 'rt') as f:
    body = f.read()

print 'FILE:'
print body
```

运行脚本之后，日志消息写至 logging_example.out。

```
$ python logging_file_example.py
```

```
FILE:
DEBUG:root:This message should go to the log file
```


14.9.3 旋转日志文件

反复运行这个脚本会将更多消息追加到这个文件。要想每次程序运行时创建一个新文件，可以向 `basicConfig()` 的参数 `filemode` 传入值 “w”。不过，最好不要采用这种方式管理文件的创建，更好的做法是使用一个 `RotatingFileHandler`，它会自动创建新文件，同时保留原来的日志文件。

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)
# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(LOG_FILENAME,
                                                maxBytes=20,
                                                backupCount=5,
                                                )

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)
for filename in logfiles:
    print filename
```

最后会得到 6 个单独的文件，分别包含应用的部分日志历史。

```
$ python logging_rotatingfile_example.py
```

```
logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

最新的文件总是 `logging_rotatingfile_example.out`，每次达到大小限制时，就会加后缀 .1 重命名。现有的各个备份文件也会重命名，使后缀递增（.1 变成 .2，等等），.5 文件会被删除。

注意：显然，这个例子将日志长度设置得太小，这只是一个极端的例子。在实际程序中要把

`maxBytes` 设置为一个更合适的值。

14.9.4 详细级别

`logging` API 还有一个有用的特性，能够采用不同的日志级别（log level）生成不同的消息。例如，这说明代码可以附带调试消息，可以适当地设置日志级别，从而不会在生产系统中写出这些调试消息。表 14.2 列出了 `logging` 定义的日志级别。

表 14.2 日志级别

级 别	值
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
UNSET	0

对于某个级别的日志消息，只有当处理器和日志记录器配置为可以发布该级别（或更高级别）的消息时，才会发布这个日志消息。例如，如果一个消息的级别是 `CRITICAL`，而日志记录器设置为 `ERROR`，这个消息就会发出（ $50 > 40$ ）。如果消息是 `WARNING`，而日志记录器设置为只生成设置为 `ERROR` 的消息，这个消息就不会发出（ $30 < 40$ ）。

```
import logging
import sys

LEVELS = { 'debug':logging.DEBUG,
            'info':logging.INFO,
            'warning':logging.WARNING,
            'error':logging.ERROR,
            'critical':logging.CRITICAL,
            }

if len(sys.argv) > 1:
    level_name = sys.argv[1]
    level = LEVELS.get(level_name, logging.NOTSET)
    logging.basicConfig(level=level)

    logging.debug('This is a debug message')
    logging.info('This is an info message')
    logging.warning('This is a warning message')
    logging.error('This is an error message')
    logging.critical('This is a critical error message')
```



运行这个脚本并提供参数（如“debug”或“warning”），查看在不同级别会显示哪些消息。

```
$ python logging_level_example.py debug
DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message

$ python logging_level_example.py info

INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message
```

14.9.5 命名日志记录器实例

前面的所有日志消息都内嵌有“根”。logging 模块支持一个日志记录器的层次结构，各个日志记录器有不同的名字。要指出一个特定的日志消息来自哪里，一种容易的方法是对各个模块使用一个单独的日志记录器对象。每个新的日志记录器会从其父对象继承配置，发送到一个日志记录器的日志消息会包含这个日志记录器的名。每个日志记录器可以采用不同的方式配置（这是可选的），从而以不同的方式处理来自不同模块的消息。下面的例子展示了如何记录来自不同模块的日志，以便跟踪消息的来源。

```
import logging

logging.basicConfig(level=logging.WARNING)

logger1 = logging.getLogger('package1.module1')
logger2 = logging.getLogger('package2.module2')

logger1.warning('This message comes from one module')
logger2.warning('And this message comes from another module')
```

输出如下：

```
$ python logging_modules_example.py

WARNING:package1.module1:This message comes from one module
WARNING:package2.module2:And this message comes from another module
```

还有更多选项来配置日志记录，包括不同的日志消息格式化选项、将消息传送到多个目标，以及使用一个套接字接口动态改变一个长时间运行的应用的配置。所有这些选项都在库模块文档中做了详细介绍。

参见：

logging (<http://docs.python.org/library/logging.html>) 这个模块的标准库文档。

14.10 fileinput——命令行过滤器框架

作用：创建命令行过滤器程序，以处理来自输入流的文本行。

Python 版本：1.5.2 及以后版本

fileinput 模块是一个框架，可以用来创建命令行程序作为过滤器处理文本文件。

14.10.1 M3U 文件转换为 RSS

过滤器的一个例子是 m3utorss，这个程序可以将一组 MP3 文件转换为一个可以作为播客共享的 RSS 提要。程序的输入是一个或多个 m3u 文件，其中列出要发布的 MP3 文件。输出是一个打印到控制台的 RSS 提要。要处理输入，程序需要迭代处理文件名列表，并完成以下工作：

- 打开各个文件。
- 读取文件的各行。
- 明确这一行是否指示一个 MP3 文件。
- 如果是，则从 mp3 文件中抽取 RSS 提要所需的信息。
- 打印输出。

所有这些文件处理都可以手工编写代码完成。这并不太复杂，而且通过一些测试，甚至错误处理也可以自行编写。不过 fileinput 可以处理所有这些细节，使程序大为简化。

```
for line in fileinput.input(sys.argv[1:]):
    mp3filename = line.strip()
    if not mp3filename or mp3filename.startswith('#'):
        continue
    item = SubElement(rss, 'item')
    title = SubElement(item, 'title')
    title.text = mp3filename
    encl = SubElement(item, 'enclosure',
                      {'type': 'audio/mpeg',
                       'url': mp3filename})
```

input() 函数取要检查的文件名列表作为参数。如果这个列表为空，模块会从标准输入读取数据。这个函数会返回一个迭代器，由所处理的文本文件生成各个文本行。调用者只需循环处理各行，跳过空格和注释，查找指向 MP3 文件的引用。

以下是完整的程序。

```
import fileinput
import sys
import time
from xml.etree.ElementTree import Element, SubElement, tostring
from xml.dom import minidom

# Establish the RSS and channel nodes
rss = Element('rss', {'xmlns:dc': "http://purl.org/dc/elements/1.1/",
                      'version': '2.0',
```

```

    })
    channel = SubElement(rss, 'channel')
    title = SubElement(channel, 'title')
    title.text = 'Sample podcast feed'
    desc = SubElement(channel, 'description')
    desc.text = 'Generated for PyMOTW'
    pubdate = SubElement(channel, 'pubDate')
    pubdate.text = time.asctime()
    gen = SubElement(channel, 'generator')
    gen.text = 'http://www.doughellmann.com/PyMOTW/'

    for line in fileinput.input(sys.argv[1:]):
        mp3filename = line.strip()
        if not mp3filename or mp3filename.startswith('#'):
            continue
        item = SubElement(rss, 'item')
        title = SubElement(item, 'title')
        title.text = mp3filename
        encl = SubElement(item, 'enclosure',
                           {'type': 'audio/mpeg',
                            'url': mp3filename})
    rough_string = tostring(rss)
    reparsed = minidom.parseString(rough_string)
    print reparsed.toprettyxml(indent="  ")

```

这个示例输入文件包含多个 MP3 文件的文件名。

```

# This is a sample m3u file
episode-one.mp3
episode-two.mp3

```

利用以上示例输入，运行 `fileinput_example.py` 可以生成 RSS 格式的 XML 数据。

```
$ python fileinput_example.py sample_data.m3u
```

```

<?xml version="1.0" ?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <title>
      Sample podcast feed
    </title>
    <description>
      Generated for PyMOTW
    </description>
    <pubDate>
      Sun Nov 28 22:55:09 2010
    </pubDate>
    <generator>
      http://www.doughellmann.com/PyMOTW/

```

```

    </generator>
</channel>
<item>
    <title>
        episode-one.mp3
    </title>
    <enclosure type="audio/mpeg" url="episode-one.mp3"/>
</item>
<item>
    <title>
        episode-two.mp3
    </title>
    <enclosure type="audio/mpeg" url="episode-two.mp3"/>
</item>
</rss>

```

14.10.2 进度元数据

在前面的例子中，文件名和正在处理的行号并不重要。但其他工具（如类 `grep` 的搜索）可能需要这个信息。`fileinput` 包含一些函数来访问有关当前行的所有元数据（`filename()`、`filelineno()` 和 `lineno()`）。

```

import fileinput
import re
import sys

pattern = re.compile(sys.argv[1])

for line in fileinput.input(sys.argv[2:]):
    if pattern.search(line):
        if fileinput.isstdin():
            fmt = '{lineno}:{line}'
        else:
            fmt = '{filename}:{lineno}:{line}'
        print fmt.format(filename=fileinput.filename(),
                        lineno=fileinput.filelineno(),
                        line=line.rstrip())

```

可以用一个基本的模式匹配循环来查找串“`fileinput`”在这些示例源文件中的出现。

```
$ python fileinput_grep.py fileinput *.py
```

```

fileinput_change_subnet.py:10:import fileinput
fileinput_change_subnet.py:17:for line in fileinput.input(files, inp
lace=True):
fileinput_change_subnet_noisy.py:10:import fileinput
fileinput_change_subnet_noisy.py:18:for line in fileinput.input(file
s, inplace=True):
fileinput_change_subnet_noisy.py:19:    if fileinput.isfirstline():

```

```

fileinput_change_subnet_noisy.py:21:                                     fileinp
ut.filename())
fileinput_example.py:6: """Example for fileinput module.
fileinput_example.py:10: import fileinput
fileinput_example.py:30: for line in fileinput.input(sys.argv[1:]):
fileinput_grep.py:10: import fileinput
fileinput_grep.py:16: for line in fileinput.input(sys.argv[2:]):
fileinput_grep.py:18:         if fileinput.isstdin():
fileinput_grep.py:22:         print fmt.format(filename=fileinput.fil
ename(),
fileinput_grep.py:23:                                     lineno=fileinput.filel
ineno(),

```

还可以从标准输入读取文本。

```
$ cat *.py | python fileinput_grep.py fileinput
```

```

10: import fileinput
17: for line in fileinput.input(files, inplace=True):
29: import fileinput
37: for line in fileinput.input(files, inplace=True):
38:     if fileinput.isfirstline():
40:         fileinput.filename()
54: """Example for fileinput module.
58: import fileinput
78: for line in fileinput.input(sys.argv[1:]):
101: import fileinput
107: for line in fileinput.input(sys.argv[2:]):
109:     if fileinput.isstdin():
113:         print fmt.format(filename=fileinput.filename(),
114:                             lineno=fileinput.filelineno(),

```

14.10.3 原地过滤

另一种常见的文件处理操作是原地（in-place）修改一个文件的内容。例如，如果一个子网范围改变，UNIX 主机文件就可能需要更新。

```

##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
##
127.0.0.1        localhost
255.255.255.255 broadcasthost
::1             localhost
fe80::1%lo0     localhost
10.16.177.128   hubert hubert.hellfly.net
10.16.177.132   cubert cubert.hellfly.net
10.16.177.136   zoidberg zoidberg.hellfly.net

```

要自动完成这个修改，安全的做法是根据输入创建一个新文件，然后用编辑后的副本替换原来的文件。fileinput 使用 inplace 选项自动支持这个方法。

```
import fileinput
import sys

from_base = sys.argv[1]
to_base = sys.argv[2]
files = sys.argv[3:]

for line in fileinput.input(files, inplace=True):
    line = line.rstrip().replace(from_base, to_base)
    print line
```

尽管脚本使用了 print，但是由于 fileinput 将标准输出重定向到所覆盖的文件，所以不生成任何输出。

```
$ python fileinput_change_subnet.py 10.16. 10.17. etc_hosts.txt
```

更新后的文件包含了 10.16.0.0/16 网络上所有服务器更改后的 IP 地址。

```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
##
127.0.0.1          localhost
255.255.255.255 broadcasthost
::1               localhost
fe80::1%lo0       localhost
10.17.177.128     hubert hubert.hellfly.net
10.17.177.132     cubert cubert.hellfly.net
10.17.177.136     zoidberg zoidberg.hellfly.net
```

处理开始之前，会使用原来的文件名加上 .bak 创建一个备份文件。

```
import fileinput
import glob
import sys

from_base = sys.argv[1]
to_base = sys.argv[2]
files = sys.argv[3:]

for line in fileinput.input(files, inplace=True):
    if fileinput.isfirstline():
        sys.stderr.write('Started processing %s\n' %
                          fileinput.filename())
        sys.stderr.write('Directory contains: %s\n' %
```



```

                                glob.glob('etc_hosts.txt*'))
    line = line.rstrip().replace(from_base, to_base)
    print line

    sys.stderr.write('Finished processing\n')
    sys.stderr.write('Directory contains: %s\n' %
                     glob.glob('etc_hosts.txt*'))

```

输入结束时删除这个备份文件。

```

$ python fileinput_change_subnet_noisy.py 10.16. 10.17. etc_hosts.txt

Started processing etc_hosts.txt
Directory contains: ['etc_hosts.txt', 'etc_hosts.txt.bak']
Finished processing
Directory contains: ['etc_hosts.txt']

```

参见：

fileinput (<http://docs.python.org/library/fileinput.html>) 这个模块的标准库文档。

m3utorss (www.doughellmann.com/projects/m3utorss) 这个脚本可以将列出 MP3 的 M3U 文件转换为一个适合作播客提要的 RSS 文件。

7.6.8 节 提供了使用 ElementTree 生成 XML 的更多详细内容。

14.11 atexit——程序关闭回调

作用：注册程序关闭时调用的函数。

Python 版本：2.1.3 及以后版本

atexit 模块提供了一个接口，来注册程序正常关闭时调用的函数。sys 模块还提供了一个 hook，sys.exitfunc，不过这里只能注册一个函数。atexit 注册表可以由多个模块和库同时使用。

14.11.1 示例

下面的例子通过 register() 注册了一个函数。

```

import atexit

def all_done():
    print 'all_done()'

print 'Registering'
atexit.register(all_done)
print 'Registered'

```

由于程序不做其他事情，会立即调用 all_done()。

```
$ python atexit_simple.py
```



```

Registering
Registered
all_done()

```

还可以注册多个函数，并向注册的函数传递参数。这对于妥善地断开数据库连接、删除临时文件等等可能很有用。不用为需要释放的资源维护一个特殊的列表，完全可以对每个资源注册一个单独的清理函数。

```

import atexit

def my_cleanup(name):
    print 'my_cleanup(%s)' % name
atexit.register(my_cleanup, 'first')
atexit.register(my_cleanup, 'second')
atexit.register(my_cleanup, 'third')

```

退出函数会按注册的逆序来调用。这个方法以模块导入顺序（相应地，也就是注册其 `atexit` 函数的顺序）的逆序完成模块清理，这会减少依赖冲突。

```

$ python atexit_multiple.py

my_cleanup(third)
my_cleanup(second)
my_cleanup(first)

```

14.11.2 什么情况下不调用 `atexit` 函数

如果满足以下任何一个条件，就不会调用为 `atexit` 注册的回调。

- 程序由于一个信号而中止。
- 直接被调用了 `os._exit()`。
- 检测到解释器中的一个致命错误。

可以更新 `subprocess` 一节的例子，显示程序因为一个信号中止时发生了什么。这里涉及两个文件，父程序和子程序。父程序启动子程序，暂停，然后将其中止。

```

import os
import signal
import subprocess
import time

proc = subprocess.Popen('atexit_signal_child.py')
print 'PARENT: Pausing before sending signal...'
time.sleep(1)
print 'PARENT: Signaling child'
os.kill(proc.pid, signal.SIGTERM)

```

子程序建立一个 `atexit` 回调，然后休眠，直至信号到来。

```

import atexit

```

```

import time
import sys
def not_called():
    print 'CHILD: atexit handler should not have been called'

print 'CHILD: Registering atexit handler'
sys.stdout.flush()
atexit.register(not_called)

print 'CHILD: Pausing to wait for signal'
sys.stdout.flush()
time.sleep(5)

```

运行时，输出如下：

```

$ python atexit_signal_parent.py

CHILD: Registering atexit handler
CHILD: Pausing to wait for signal
PARENT: Pausing before sending signal...
PARENT: Signaling child

```

子程序不会打印嵌在 `not_called()` 中的消息。

如果程序使用了 `os._exit()`，就不会再调用 `atexit` 回调。

```

import atexit
import os

def not_called():
    print 'This should not be called'

print 'Registering'
atexit.register(not_called)
print 'Registered'

print 'Exiting...'
os._exit(0)

```

由于这个例子绕过了正常的退出路径，所以没有运行回调。

```
$ python atexit_os_exit.py
```

要确保回调运行，可以在语句外运行来执行或者调用 `sys.exit()` 使程序中止。

```

import atexit
import sys

def all_done():
    print 'all_done()'

print 'Registering'

```

```

atexit.register(all_done)
print 'Registered'

print 'Exiting...'
sys.exit()

```

这个例子调用了 `sys.exit()`，所以会调用注册的回调。

```
$ python atexit_sys_exit.py
```

```

Registering
Registered
Exiting...
all_done()

```

14.11.3 处理异常

`atexit` 回调中所产生异常的 `Traceback` 会打印到控制台上，最后产生的异常会重新抛出，作为程序的最后一个错误消息。

```

import atexit

def exit_with_exception(message):
    raise RuntimeError(message)

atexit.register(exit_with_exception, 'Registered first')
atexit.register(exit_with_exception, 'Registered second')

```

注册顺序会控制执行顺序。如果一个回调中的某个错误引入了另一个回调中的一个错误（越先注册，越后调用），最后的错误消息可能并不是为用户显示的最有用的错误消息。

```
$ python atexit_exception.py
```

```

Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "atexit_exception.py", line 37, in exit_with_exception
    raise RuntimeError(message)
RuntimeError: Registered second
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "atexit_exception.py", line 37, in exit_with_exception
    raise RuntimeError(message)
RuntimeError: Registered first

```

```
Error in sys.exitfunc:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/atexit.py", line 24, in _run_exitfuncs
    func(*targs, **kargs)
  File "atexit_exception.py", line 37, in exit_with_exception
    raise RuntimeError(message)
RuntimeError: Registered first
```

通常最好在清理函数中处理异常并悄悄地记入日志，因为程序退出时显示一大堆错误会显得很乱。

参见：

atexit (<http://docs.python.org/library/atexit.html>) 这个模块的标准库文档。

14.12 sched——定时事件调度器

作用：通用事件调度器。

Python 版本：1.4 及以后版本

sched 模块实现了一个通用事件调度器，可以在指定时刻运行任务。调度器类使用一个时间（time）函数来掌握当前时间，另外利用一个延迟（delay）函数等待一个指定时段。具体的时间单位并不重要，从而使接口足够灵活，可以用于实现很多用途。

调用 time 函数时不带任何参数，它会返回一个表示当前时间的数。调用 delay 函数要提供一个整数参数，使用的单位与时间函数相同，返回之前会等待指定数目的时间单位。例如，time.time() 和 time.sleep() 函数就满足这些需求。

要支持多线程应用，生成各事件之后可以调用延迟函数并提供参数 0，确保其他线程也有机会运行。

14.12.1 有延迟地运行事件

可以调度事件在一个延迟之后运行或在指定时间运行。要调度事件有一个延迟，可以使用 enter() 方法，它有 4 个参数：

- 表示延迟的数
- 优先级值
- 要调用的函数
- 函数参数的元组

这个例子调度两个不同的事件分别在 2 秒和 3 秒后运行。一旦达到事件的时间，会调用 print_event()，并打印当前时间和传至事件的 name 参数。

```
import sched
import time
```

```

scheduler = sched.scheduler(time.time, time.sleep)

def print_event(name, start):
    now = time.time()
    elapsed = int(now - start)
    print 'EVENT: %s elapsed=%s name=%s' % (time.ctime(now),
                                           elapsed,
                                           name)

start = time.time()
print 'START:', time.ctime(start)
scheduler.enter(2, 1, print_event, ('first', start))
scheduler.enter(3, 1, print_event, ('second', start))

scheduler.run()

```

运行这个程序会生成以下输出:

```

$ python sched_basic.py

START: Sun Oct 31 20:48:47 2010
EVENT: Sun Oct 31 20:48:49 2010 elapsed=2 name=first
EVENT: Sun Oct 31 20:48:50 2010 elapsed=3 name=second

```

为第一个事件打印的时间是距开始时间 2 秒, 第二个事件的时间则为距开始时间 3 秒。

14.12.2 重叠事件

run() 调用会阻塞, 直至所有事件都已经处理。每个事件都在相同的线程中运行, 所以如果一个事件需要很长时间运行, 超出了事件之间的延迟, 就会出现重叠。这个重叠可以通过推迟后面的事件来解决。这样不会丢失事件, 不过有些事件可能比其调度时间更晚调用。在下面的例子中, long_event() 会调用 sleep 休眠, 不过也可以通过完成一个复杂的计算或者是 I/O 阻塞来轻松延迟。

```

import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)

def long_event(name):
    print 'BEGIN EVENT :', time.ctime(time.time()), name
    time.sleep(2)
    print 'FINISH EVENT:', time.ctime(time.time()), name

print 'START:', time.ctime(time.time())
scheduler.enter(2, 1, long_event, ('first',))
scheduler.enter(3, 1, long_event, ('second',))

```

```
scheduler.run()
```

其结果是第一个事件一旦完成就会立即运行第二个事件，因为第一个事件花费的时间足够长，使时钟超过了第二个事件期望的开始时间。

```
$ python sched_overlap.py
```

```
START: Sun Oct 31 20:48:50 2010
BEGIN EVENT : Sun Oct 31 20:48:52 2010 first
FINISH EVENT: Sun Oct 31 20:48:54 2010 first
BEGIN EVENT : Sun Oct 31 20:48:54 2010 second
FINISH EVENT: Sun Oct 31 20:48:56 2010 second
```

14.12.3 事件优先级

如果调度多个事件在同一时间运行，就要使用事件的优先级来确定它们以何种顺序运行。

```
import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)

def print_event(name):
    print 'EVENT:', time.ctime(time.time()), name

now = time.time()
print 'START:', time.ctime(now)
scheduler.enterabs(now+2, 2, print_event, ('first',))
scheduler.enterabs(now+2, 1, print_event, ('second',))

scheduler.run()
```

这个例子需要确保事件调度为在完全相同的时间运行，所以要使用 `enterabs()` 方法而不是 `enter()`。`enterabs()` 的第一个参数是运行事件的时间，而不是要延迟的时间。

```
$ python sched_priority.py
```

```
START: Sun Oct 31 20:48:56 2010
EVENT: Sun Oct 31 20:48:58 2010 second
EVENT: Sun Oct 31 20:48:58 2010 first
```

14.12.4 取消事件

`enter()` 和 `enterabs()` 都会返回事件的一个引用，以后可以用这个引用来取消事件。由于 `run()` 阻塞，必须在一个不同的线程中取消这个事件。对于这个例子，线程开始运行调度器，并用主处理线程来取消事件。

```
import sched
import threading
import time
```

```
scheduler = sched.scheduler(time.time, time.sleep)

# Set up a global to be modified by the threads
counter = 0

def increment_counter(name):
    global counter
    print 'EVENT:', time.ctime(time.time()), name
    counter += 1
    print 'NOW:', counter

print 'START:', time.ctime(time.time())
e1 = scheduler.enter(2, 1, increment_counter, ('E1',))
e2 = scheduler.enter(3, 1, increment_counter, ('E2',))

# Start a thread to run the events
t = threading.Thread(target=scheduler.run)
t.start()

# Back in the main thread, cancel the first scheduled event.
scheduler.cancel(e1)

# Wait for the scheduler to finish running in the thread
t.join()

print 'FINAL:', counter
```

这里调度了两个事件，不过第一个事件随后被取消了。只运行了第二个事件，所以 counter 变量只递增一次。

```
$ python sched_cancel.py
```

```
START: Sun Oct 31 20:48:58 2010
EVENT: Sun Oct 31 20:49:01 2010 E2
NOW: 1
FINAL: 1
```

参见:

`sched` (<http://docs.python.org/lib/module-sched.html>) 这个模块的标准库文档。

`time` (4.1 节) `time` 模块。



Python 提供了两个模块来支持应用处理多种自然语言和文化设置。`gettext` 用于采用不同语言创建消息编目，从而能以用户能够理解的语言显示提示语和错误消息。`locale` 会考虑到文化差异，改变数字、货币、日期和时间的格式化方式，如如何指示负数，本地货币符号是什么等等。这两个模块都会与其他工具和操作环境交互，使 Python 应用能够与系统上的其他程序很好地配合。

15.1 `gettext`——消息编目

作用：完成国际化的消息编目 API。

Python 版本：2.1.3 及以后版本

`gettext` 模块提供了一个纯 Python 实现，与 GNU `gettext` 库兼容，用于完成消息转换和编目管理。利用 Python 源代码发布版提供的工具，可以从一组源文件中抽取消息，构建一个包含转换的消息编目，并使用这个消息编目在运行时为用户显示一个适当的消息。

消息编目可以用于为程序提供国际化接口，使用适合用户的语言来显示消息。还可以用于其他消息定制，包括为不同包装器或合作伙伴的界面“换肤”。

注意：尽管标准库文档声称 Python 已经包含所有必要的工具，但是即使提供了适当的命令行参数，`pygettext.py` 也无法抽取包装在 `ungettext` 调用中的消息。实际上，这些例子使用了 GNU `gettext` 工具集的 `xgettext`。

15.1.1 转换 workflow 概述

建立和使用转换的过程包括 5 个步骤。

(1) 标识并标记源代码中包含待转换消息的字面量串。

首先在程序源代码中标识需要转换的消息，并标记字面量串，以便抽取程序发现这些字面量串。

(2) 抽取消息。

标识源代码中可转换的串之后，使用 `xgettext` 抽取出这些串，并创建一个 `.pot` 文件或转换模板 (translation template)。这个模板是一个文本文件，包含标识的所有串的副本及对应其转换的占位符。

(3) 转换消息。

将 .pot 文件的一个副本提供给转换器，将扩展名改为 .po。这个 .po 文件是一个可编辑的源文件，用作下一步编译的输入。转换器要更新这个文件中的首部文本，提供所有串的连接。

(4) 由转换“编译”消息编目。

转换器发回完整的 .po 文件后，使用 msgfmt 将这个文本文件编译为二进制编目格式。运行时编目查找代码将使用这个二进制格式。

(5) 运行时加载并启动适当的消息编目。

最后一步是向应用添加几行代码，配置和加载消息编目，并安装转换函数。对此有几种方法，这些方法各有优缺点。

本节余下的内容将更详细地介绍这些步骤，首先从需要完成的代码修改开始。

15.1.2 由源代码创建消息编目

gettext 首先在一个转换数据库中查找字面量串，并取出适当的转换串。访问这个编目的函数有很多变种，取决于字符串是否为 Unicode 编码。常用模式是将适当的查找函数与名 “_”（单个下划线字符）绑定，使得代码中不会堆积大量长名函数调用。

消息抽取程序 xgettext 会查找嵌入在编目查找函数（catalog lookup function）调用中的消息。它知道不同的源语言，并分别使用适当的解析器。如果查找函数有别名，或者增加了额外的函数，要为 xgettext 提供这些额外符号的名称，从而在抽取消息时能够考虑到。

以下脚本提供了一个消息，可以完成转换。

```
import gettext

# Set up message catalog access
t = gettext.translation('example', 'locale', fallback=True)
_ = t.ugettext

print _('This message is in the script.')
```

这个例子使用了查找函数的 Unicode 版本 ugettext()。文本 “This message is in the script.” 是将来由编目替换的消息。这里启用了 Fallback 模式，所以如果运行脚本时没有一个消息编目，则会打印内联的消息。

```
$ python gettext_example.py
```

```
This message is in the script.
```

下一步是抽取消息，并创建 .pot 文件，这里可以使用 Python 的 pygettext.py 或 GNU 工具 xgettext。

```
$ xgettext -o example.pot gettext_example.py
```

生成的输出文件包含以下内容：

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license
# as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2010-11-28 23:16-0500\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: gettext_example.py:16
msgid "This message is in the script."
msgstr ""
```

消息编目要安装到按域 (domain) 和语言 (language) 组织的目录中。域通常是一个惟一值, 如应用名。在这里, 域是 `gettext_example`。语言值则由用户环境在运行时通过某个环境变量 (`LANGUAGE`、`LC_ALL`、`LC_MESSAGES` 或 `LANG`) 提供, 这取决于其配置和平台。这些例子运行时都将语言设置为 `en_US`。

模板已经准备好, 下一步是创建必要的目录结构, 并把模板复制到适当的位置。PyMOTW 源码树中的 `locale` 目录可以作为这些示例消息编目目录的根, 不过通常最好使用全系统都可以访问的一个目录, 从而使所有用户都能访问消息编目。这个编目输入源文件的完整路径为 `$localedir/$language/LC_MESSAGES/$domain.po`, 实际编目的文件扩展名为 `.mo`。

将 `example.pot` 复制到 `locale/en_US/LC_MESSAGES/example.po`, 编辑这个文件, 改变首部中的值, 并设置替换消息, 从而创建编目。结果如下所示。

```
# Messages from gettext_example.py.
# Copyright (C) 2009 Doug Hellmann
# Doug Hellmann <doug.hellmann@gmail.com>, 2009.
#
msgid ""
msgstr ""
"Project-Id-Version: PyMOTW 1.92\n"
"Report-Msgid-Bugs-To: Doug Hellmann <doug.hellmann@gmail.com>\n"
"POT-Creation-Date: 2009-06-07 10:31+EDT\n"
"PO-Revision-Date: 2009-06-07 10:31+EDT\n"
"Last-Translator: Doug Hellmann <doug.hellmann@gmail.com>\n"
```

```
"Language-Team: US English <doug.hellmann@gmail.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

```
#: gettext_example.py:16
msgid "This message is in the script."
msgstr "This message is in the en_US catalog."
```

使用 `msgformat` 从 `.po` 文件构建这个编目。

```
$ cd locale/en_US/LC_MESSAGES/; msgfmt -o example.mo example.po
```

现在运行脚本时会打印编目中的消息而不是内联字符串。

```
$ python gettext_example.py
```

```
This message is in the en_US catalog.
```

15.1.3 运行时查找消息编目

如前所述，包含消息编目的 `locale` 目录根据语言来组织，编目按程序的域命名。不同的操作系统分别定义了自己的默认值，不过 `gettext` 并不知道所有这些默认值。它使用一个默认的 `locale` 目录 `sys.prefix + '/share/locale'`，但是大多数情况下，更安全的做法是显式地提供一个 `localedir` 值而不是依赖于这个默认值有效。`find()` 函数负责在运行时找到一个合适的消息编目。

```
import gettext
```

```
catalogs = gettext.find('example', 'locale', all=True)
print 'Catalogs:', catalogs
```

路径的语言部分 (`language`) 由某个环境变量 (`LANGUAGE`、`LC_ALL`、`LC_MESSAGES` 和 `LANG`) 得到，这些环境变量可以用于配置本地化特性。总会使用第一个已设置的变量。通过将值用冒号 (:) 分隔，可以选择多种语言。要看这是如何做到的，可以再使用另一个消息编目做一些试验。

```
$ (cd locale/en_CA/LC_MESSAGES/; msgfmt -o example.mo example.po)
$ python gettext_find.py
```

```
Catalogs: ['locale/en_US/LC_MESSAGES/example.mo']
```

```
$ LANGUAGE=en_CA python gettext_find.py
```

```
Catalogs: ['locale/en_CA/LC_MESSAGES/example.mo']
```

```
$ LANGUAGE=en_CA:en_US python gettext_find.py
```

```
Catalogs: ['locale/en_CA/LC_MESSAGES/example.mo',
```

```
'locale/en_US/LC_MESSAGES/example.mo']

$ LANGUAGE=en_US:en_CA python gettext_find.py

Catalogs: ['locale/en_US/LC_MESSAGES/example.mo',
'locale/en_CA/LC_MESSAGES/example.mo']
```

尽管 find() 显示了完整的编目列表, 不过实际上只加载了这个序列中的第一个编目来完成消息查找。

```
$ python gettext_example.py

This message is in the en_US catalog.

$ LANGUAGE=en_CA python gettext_example.py

This message is in the en_CA catalog.

$ LANGUAGE=en_CA:en_US python gettext_example.py

This message is in the en_CA catalog.

$ LANGUAGE=en_US:en_CA python gettext_example.py

This message is in the en_US catalog.
```

15.1.4 复数值

简单的消息替换可以处理大多数转换需求, 不过 gettext 将复数处理为一种特殊情况。一个消息的单数和复数形式之间会有差别, 而且根据具体语言这种差别可能也有所不同, 有些只是单词末尾不同, 有些则是整个句子结构都不同。根据复数的层次, 可能还会有不同的形式。为了更容易地管理复数 (在某些情况下, 甚至是为了能管理复数), 模块提供了一组单独的函数来询问一个消息的复数形式。

```
from gettext import translation
import sys

t = translation('gettext_plural', 'locale', fallback=True)
num = int(sys.argv[1])
msg = t.ungettext('%(num)d means singular.',
                  '%(num)d means plural.',
                  num)

# Still need to add the values to the message ourself.
print msg % {'num':num}
```

使用 ungettext() 来访问一个消息的复数形式的 Unicode 版本。参数是要转换的消息和项数。

```
$ xgettext -L Python -o plural.pot gettext_plural.py
```

由于转换的候选形式有多种，这些替换形式会列在一个数组中。通过使用数组，就可以对有多种复数形式的语言完成转换（例如，波兰语就用不同的形式来表示相对数量）。

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license
# as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2010-11-28 23:09-0500\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"

#: gettext_plural.py:15
#, python-format
msgid "%(num)d means singular."
msgid_plural "%(num)d means plural."
msgstr[0] ""
msgstr[1] ""
```

除了填入转换串之外，还需要告诉库采用哪种复数形式，使它知道对应给定的数量值如何在数组中索引。行“Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n”包含两个值，需要手工替换。nplurals 是一个整数，指示数组的大小（使用的转换数），plural 是一个 C 语言表达式，用于将得到的数量转换为查找转换时需要的数组索引。字面量串 n 会替换为传递给 ungettext() 的数量。

例如，英语包括两种复数形式。数量 0 会处理为复数（“0 bananas”）。这是 Plural-Forms 项。

```
Plural-Forms: nplurals=2; plural=n != 1;
```

单数转换则在位置 0，复数转换在位置 1。

```
# Messages from gettext_plural.py
# Copyright (C) 2009 Doug Hellmann
# This file is distributed under the same license
# as the PyMOTW package.
```

```
# Doug Hellmann <doug.hellmann@gmail.com>, 2009.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PyMOTW 1.92\n"
"Report-Msgid-Bugs-To: Doug Hellmann <doug.hellmann@gmail.com>\n"
"POT-Creation-Date: 2009-06-14 09:29-0400\n"
"PO-Revision-Date: 2009-06-14 09:29-0400\n"
"Last-Translator: Doug Hellmann <doug.hellmann@gmail.com>\n"
"Language-Team: en_US <doug.hellmann@gmail.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=2; plural=n != 1;"

#: gettext_plural.py:15
#, python-format
msgid "%(num)d means singular."
msgid_plural "%(num)d means plural."
msgstr[0] "In en_US, %(num)d is singular."
msgstr[1] "In en_US, %(num)d is plural."
```

编译编目之后，运行几次测试脚本，展示不同的 *n* 值如何转换为对应转换字符串的索引。

```
$ cd locale/en_US/LC_MESSAGES/; msgfmt -o plural.mo plural.po
$ python gettext_plural.py 0
```

```
0 means plural.
```

```
$ python gettext_plural.py 1
```

```
1 means singular.
```

```
$ python gettext_plural.py 2
```

```
2 means plural.
```

15.1.5 应用与模块本地化

转换的作用域定义了如何安装 `gettext`，以及如何用于一个代码体。

应用本地化

对于全应用范围的转换，可以让作者使用 `__builtins__` 命名空间全局地安装类似 `ungettext()` 的函数，这是可以接受的，因为它们可以在应用代码的顶层加以控制。

```
import gettext
gettext.install('gettext_example', 'locale',
                unicode=True, names=['ungettext'])
```

```
print _('This message is in the script.')
```

`install()` 函数将 `gettext()` 绑定到 `__builtins__` 命名空间中的名 `_()`。它还增加了 `ngettext()` 和 `names` 中列出的其他函数。如果 `unicode` 为 `true`，将使用函数的 Unicode 版本而不是默认的 ASCII 版本。

模块本地化

对于一个库或单个模块，修改 `__builtins__` 并不是一个好的想法，因为这可能引入与某个应用全局值的冲突。相反，应当在模块前面手工地导入或重新绑定转换函数名。

```
import gettext
t = gettext.translation('gettext_example', 'locale', fallback=True)
_ = t.ugettext
ngettext = t.ungettext

print _('This message is in the script.')
```

15.1.6 切换转换

前面的例子在整个程序期间都使用了一个转换。有些情况下，特别是对于 Web 应用，还需要在不同时间使用不同的消息编目，而不是退出和重新设置环境。对于这些情况，`gettext` 中提供的基于类的 API 会更为方便。这些 API 调用实际上与这一节中介绍的全局调用相同，不过会发布消息编目对象，而且可以直接管理，因此可以使用多个编目。

参见：

`gettext` (<http://docs.python.org/library/gettext.html>) 这个模块的标准库文档。

`locale` (15.2 节) 其他本地化工具。

GNU `gettext` (www.gnu.org/software/gettext/) 这个模块的消息编目格式、API 等等都基于 GNU 原来的 `gettext` 包。编目文件格式是可兼容的，命令行脚本有类似的选项（甚至完全相同）。GNU `gettext` 手册 (www.gnu.org/software/gettext/manual/gettext.html) 中对文件格式做了详细的描述，并介绍了处理这些文件格式的 GNU 版本工具。

Plural forms (www.gnu.org/software/gettext/manual/gettext.html#Plural-forms) 处理不同语言中单词和句子的复数形式。

Internationalizing Python (www.python.org/workshops/1997-10/proceedings/loewis.html) Martin von Löwis 撰写的关于 Python 应用国际化技术的一篇文章。

Django Internationalization (<http://docs.djangoproject.com/en/dev/topics/i18n/>) 另一个关于使用 `gettext` 的很好的信息来源，还包括一些真实的示例。

15.2 locale——文化本地化 API

作用：格式化和解析依赖于位置或语言的值。

Python 版本：1.5 及以后版本

`locale` 模块是 Python 国际化和本地化支持库的一部分。它提供了一种标准方法，来处理可能

依赖于用户语言或位置的操作。例如，它会处理如何将数字格式化为货币形式，比较字符串来完成排序，以及处理日期。它不涉及转换（见 `gettext` 模块）或 Unicode 编码（见 `codecs` 模块）。

注意：改变本地化环境可能会给整个应用带来影响，所以推荐的实践做法是避免改变库中的值，而是让应用一次性设置。本节的例子会在一个小程序中多次改变本地化环境，以强调不同本地化环境设置的差别。通常更合理的做法是由应用在启动时设置一次本地化环境，然后就不再改变。

本节将介绍 `locale` 模块中的一些高级函数。其他函数则为低级函数（`format_string()`）或者与管理应用的本地化环境有关（`resetlocale()`）。

15.2.1 探查当前本地化环境

要让用户改变应用的本地化环境设置，最常见的方式是通过一个环境变量（`LC_ALL`、`LC_CTYPE`、`LANG` 或 `LANGUAGE`，这取决于使用哪个平台）。然后应用调用 `setlocale()` 而不是指定硬编码的值，并使用环境值。

```
import locale
import os
import pprint
import codecs
import sys

sys.stdout = codecs.getwriter('UTF-8')(sys.stdout)

# Default settings based on the user's environment.
locale.setlocale(locale.LC_ALL, '')

print 'Environment settings:'
for env_name in [ 'LC_ALL', 'LC_CTYPE', 'LANG', 'LANGUAGE' ]:
    print '\t%s = %s' % (env_name, os.environ.get(env_name, ''))

# What is the locale?
print
print 'Locale from environment:', locale.getlocale()

template = """
Numeric formatting:

    Decimal point      : "%(decimal_point)s"
    Grouping positions : %(grouping)s
    Thousands separator: "%(thousands_sep)s"

Monetary formatting:
```

```

International currency symbol      : "%(int_curr_symbol)r"
Local currency symbol              : %(currency_symbol)r
Unicode version                    : %(currency_symbol_u)s
Symbol precedes positive value    : %(p_cs_precedes)s
Symbol precedes negative value    : %(n_cs_precedes)s
Decimal point                      : "%(mon_decimal_point)s"
Digits in fractional values        : %(frac_digits)s
Digits in fractional values, international: %(int_frac_digits)s
Grouping positions                 : %(mon_grouping)s
Thousands separator               : "%(mon_thousands_sep)s"
Positive sign                     : "%(positive_sign)s"
Positive sign position            : %(p_sign_posn)s
Negative sign                     : "%(negative_sign)s"
Negative sign position            : %(n_sign_posn)s

sign_positions = {
    0 : 'Surrounded by parentheses',
    1 : 'Before value and symbol',
    2 : 'After value and symbol',
    3 : 'Before value',
    4 : 'After value',
    locale.CHAR_MAX : 'Unspecified',
}

info = {}
info.update(locale.localeconv())
info['p_sign_posn'] = sign_positions[info['p_sign_posn']]
info['n_sign_posn'] = sign_positions[info['n_sign_posn']]
# convert the currency symbol to unicode
info['currency_symbol_u'] = info['currency_symbol'].decode('utf-8')

print (template % info)

```

`localeconv()` 方法返回一个字典，其中包含本地化环境的约定。完整的值名和定义列表在标准库文档中已经给出。

在运行 OS X 10.6 的 Mac 上（所有变量都未设置），会生成以下输出。

```
$ export LANG=; export LC_CTYPE=; python locale_env_example.py
```

```

Environment settings:
    LC_ALL =
    LC_CTYPE =
    LANG =
    LANGUAGE =

```

```
Locale from environment: (None, None)
```

```
Numeric formatting:
```

```

Decimal point      : "."
Grouping positions : [3, 3, 0]
Thousands separator: ","

```

Monetary formatting:

```

International currency symbol : "'USD '"
Local currency symbol         : '$'
Unicode version               : $
Symbol precedes positive value : 1
Symbol precedes negative value : 1
Decimal point                 : "."
Digits in fractional values    : 2
Digits in fractional values, international: 2
Grouping positions            : [3, 3, 0]
Thousands separator           : ","
Positive sign                  : ""
Positive sign position         : Before value and symbol
Negative sign                  : "-"
Negative sign position         : Before value and symbol

```

运行同样的脚本，但设置了 LANG 变量，会显示本地化环境和默认编码如何改变。

法语 (fr_FR):

```
$ LANG=fr_FR LC_CTYPE=fr_FR LC_ALL=fr_FR python locale_env_example.py
```

Environment settings:

```

LC_ALL = fr_FR
LC_CTYPE = fr_FR
LANG = fr_FR
LANGUAGE =

```

Locale from environment: ('fr_FR', 'ISO8859-1')

Numeric formatting:

```

Decimal point      : ","
Grouping positions : [127]
Thousands separator: ""

```

Monetary formatting:

```

International currency symbol : "'EUR '"
Local currency symbol         : 'Eu'
Unicode version               : Eu
Symbol precedes positive value : 0
Symbol precedes negative value : 0
Decimal point                 : ","

```

```

Digits in fractional values          : 2
Digits in fractional values, international: 2
Grouping positions                   : [3, 3, 0]
Thousands separator                  : " "
Positive sign                        : ""
Positive sign position               : Before value and symbol
Negative sign                        : "-"
Negative sign position               : After value and symbol

```

西班牙语 (es_ES):

```
$ LANG=es_ES LC_CTYPE=es_ES LC_ALL=es_ES python locale_env_example.py
```

Environment settings:

```

LC_ALL = es_ES
LC_CTYPE = es_ES
LANG = es_ES
LANGUAGE =

```

```
Locale from environment: ('es_ES', 'ISO8859-1')
```

Numeric formatting:

```

Decimal point      : ","
Grouping positions : [127]
Thousands separator: ""

```

Monetary formatting:

```

International currency symbol : "'EUR '"
Local currency symbol          : 'Eu'
Unicode version                 Eu
Symbol precedes positive value : 1
Symbol precedes negative value : 1
Decimal point                   : ","
Digits in fractional values     : 2
Digits in fractional values, international: 2
Grouping positions              : [3, 3, 0]
Thousands separator             : "."
Positive sign                   : ""
Positive sign position          : Before value and symbol
Negative sign                   : "-"
Negative sign position          : Before value and symbol

```

葡萄牙语 (pt_PT):

```
$ LANG=pt_PT LC_CTYPE=pt_PT LC_ALL=pt_PT python locale_env_example.py
```

Environment settings:

```
LC_ALL = pt_PT
```

```
LC_CTYPE = pt_PT
LANG = pt_PT
LANGUAGE =
```

Locale from environment: ('pt_PT', 'ISO8859-1')

Numeric formatting:

```
Decimal point      : ",",
Grouping positions : []
Thousands separator: " "
```

Monetary formatting:

```
International currency symbol : "'EUR '"
Local currency symbol         : 'Eu'
Unicode version                : Eu
Symbol precedes positive value : 0
Symbol precedes negative value : 0
Decimal point                  : "."
Digits in fractional values     : 2
Digits in fractional values, international: 2
Grouping positions             : [3, 3, 0]
Thousands separator            : "."
Positive sign                   : ""
Positive sign position          : Before value and symbol
Negative sign                   : "-"
Negative sign position          : Before value and symbol
```

波兰语 (pl_PL):

```
$ LANG=pl_PL LC_CTYPE=pl_PL LC_ALL=pl_PL python locale_env_example.py
```

Environment settings:

```
LC_ALL = pl_PL
LC_CTYPE = pl_PL
LANG = pl_PL
LANGUAGE =
```

Locale from environment: ('pl_PL', 'ISO8859-2')

Numeric formatting:

```
Decimal point      : ",",
Grouping positions : [3, 3, 0]
Thousands separator: " "
```

Monetary formatting:



```

International currency symbol      : "'PLN '"
Local currency symbol              : 'z\x5c5\x82'
Unicode version                    : zł
Symbol precedes positive value    : 1
Symbol precedes negative value    : 1
Decimal point                      : ",",
Digits in fractional values        : 2
Digits in fractional values, international: 2
Grouping positions                 : [3, 3, 0]
Thousands separator                : " "
Positive sign                      : ""
Positive sign position             : After value
Negative sign                     : "- "
Negative sign position             : After value

```

15.2.2 货币

从前面的示例输出可以看到，改变本地化环境会更新货币符号设置，还会改变分隔整数和小数部分的字符。这个例子循环处理多个不同的本地化环境，针对各个本地化环境，分别打印一个格式化的正货币值和负货币值。

```

import locale

sample_locales = [ ('USA',      'en_US'),
                    ('France',   'fr_FR'),
                    ('Spain',    'es_ES'),
                    ('Portugal',  'pt_PT'),
                    ('Poland',    'pl_PL'),
                    ]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    print '%20s: %10s %10s' % (name,
                               locale.currency(1234.56),
                               locale.currency(-1234.56))

```

输出为下面这个很小的表。

```

$ python locale_currency_example.py

      USA:   $1234.56   -$1234.56
France: 1234,56 Eu  1234,56 Eu-
Spain: Eu 1234,56   -Eu 1234,56
Portugal: 1234.56 Eu -1234.56 Eu
Poland: zł 1234,56  zł 1234,56-

```

15.2.3 格式化数字

与货币无关的数字也会根据本地化环境以不同方式格式化。具体来讲，数字中会有一些分

组 (grouping) 字符, 用于将大数字分隔为可读的小块, 不同本地化环境中, 这些分组字符会改变。

```
import locale

sample_locales = [ ('USA',      'en_US'),
                    ('France',   'fr_FR'),
                    ('Spain',    'es_ES'),
                    ('Portugal', 'pt_PT'),
                    ('Poland',   'pl_PL'),
                    ]

print '%20s %15s %20s' % ('Locale', 'Integer', 'Float')
for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)

    print '%20s' % name,
    print locale.format('%15d', 123456, grouping=True),
    print locale.format('%20.2f', 123456.78, grouping=True)
```

要格式化不带货币符号的数字, 可以使用 `format()` 而不是 `currency()`。

```
$ python locale_grouping.py
```

Locale	Integer	Float
USA	123,456	123,456.78
France	123456	123456,78
Spain	123456	123456,78
Portugal	123456	123456,78
Poland	123 456	123 456,78

15.2.4 解析数字

除了以不同格式生成输出外, `locale` 模块还有助于解析输入。它包含 `atoi()` 和 `atof()` 函数, 可以根据本地化环境的数值格式约定将字符串转换为整数和浮点值。

```
import locale

sample_data = [ ('USA',      'en_US', '1,234.56'),
                 ('France',   'fr_FR', '1234,56'),
                 ('Spain',    'es_ES', '1234,56'),
                 ('Portugal', 'pt_PT', '1234.56'),
                 ('Poland',   'pl_PL', '1 234,56'),
                 ]

for name, loc, a in sample_data:
    locale.setlocale(locale.LC_ALL, loc)
    f = locale.atof(a)
    print '%20s: %9s => %f' % (name, a, f)
```

解析器会识别本地化环境的分组和小数分隔符值。

```
$ python locale_atof_example.py
```

```
USA: 1,234.56 => 1234.560000
France: 1234,56 => 1234.560000
Spain: 1234,56 => 1234.560000
Portugal: 1234.56 => 1234.560000
Poland: 1 234,56 => 1234.560000
```

15.2.5 日期和时间

本地化的另一个重要方面是日期和时间格式化。

```
import locale
import time
sample_locales = [ ('USA', 'en_US'),
                    ('France', 'fr_FR'),
                    ('Spain', 'es_ES'),
                    ('Portugal', 'pt_PT'),
                    ('Poland', 'pl_PL'),
                    ]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    format = locale.nl_langinfo(locale.D_T_FMT)
    print '%20s: %s' % (name, time.strftime(format))
```

这个例子使用本地化环境的日期格式化串来打印当前日期和时间。

```
$ python locale_date_example.py
```

```
USA: Sun Nov 28 23:53:58 2010
France: Dim 28 nov 23:53:58 2010
Spain: dom 28 nov 23:53:58 2010
Portugal: Dom 28 Nov 23:53:58 2010
Poland: ndz 28 lis 23:53:58 2010
```

参见:

locale (<http://docs.python.org/library/locale.html>) 这个模块的标准库文档。

gettext (15.1 节) 完成转换的消息编目。



第 16 章

开发工具

Python 已经演变为一个庞大的模块生态系统，这些模块的作用就是让 Python 开发人员的日子更好过，有了它们，就不再需要开发人员一切都从头开始构建。同样的道理也适用于开发人员使用的工具，即使有些工具并不真正用于程序的最后版本。本章将涵盖 Python 包含的常用模块，这些模块可以为常见的开发任务（如测试、调试和性能分析）提供便利。

对开发人员来说，最基本的帮助形式就是所使用的代码的文档。pydoc 模块可以用来从任何可移植模块源代码中包含的 docstring 生成格式化的引用文档。

Python 包含两个测试框架，可以自动地执行代码，并验证代码是否正常工作。doctest 可以从文档中包含的示例抽取测试场景，这些示例可能在源代码中，也可能作为独立的文件。unittest 是一个功能完备的自动化测试框架，支持固件、预定义测试套件和测试发现。

trace 模块会监视 Python 如何执行一个程序，生成一个报告，显示每行运行多少次。这个信息可以用于查找自动化测试套件未测试的代码路径，并研究函数调用图来查找模块之间的依赖性。

编写和运行测试可以发现大多数程序中的问题。Python 有助于更容易地完成调试，因为在大多数情况下，未处理的错误会作为 traceback 打印到屏幕上。未在一个文本控制台环境中运行程序时，可以用 traceback 为日志文件或消息对话框准备一个类似的输出。有些情况下，标准 traceback 不能提供足够的信息，对于这些情况，可以使用 cgitb 查看详细信息，如栈中每一级的局部变量设置和源代码上下文。cgitb 还可以将 traceback 格式化为 HTML，用于在 Web 应用中报告错误。

一旦找出问题的位置，可以使用 pdb 模块中的交互式调试工具单步调试代码，通过显示出错误情况的代码路径，并使用现场（live）对象和代码尝试修改，使之更容易修正。

测试并调试程序使它能正常工作之后，下一步就是研究性能。通过使用 profile 和 timeit，开发人员可以度量一个程序的运行速度，找出速度慢的部分，从而将它们隔离出来并加以改进。

运行 Python 程序时，会向解释器提供原程序源代码的一个字节编译版本。字节编译版本可以动态创建，也可以在程序打包时创建。compileall 模块提供了界面安装程序，另外还提供了打包工具，用于创建包含模块字节码的文件。可以在开发环境中使用这个模块来确定一个文件没有语法错误，并在发布程序时构建可以打包的字节编译文件。

在源代码级，pyclbr 模块提供了一个类浏览器，文本编辑器或其他程序可以使用这个类浏览器扫描 Python 源文件，查找感兴趣的符号，如函数和类，而不必导入代码，也不会潜在地触发副作用。

16.1 pydoc——模块的联机帮助

作用：从代码为 Python 模块和类生成帮助。

Python 版本：2.1 及以后版本

pydoc 模块导入了一个 Python 模块，并使用其内容在运行时生成帮助文本。只要对象中包含 docstring，输出则包括所有这些对象的 docstring，另外包括所描述模块的所有类、方法和函数。

16.1.1 纯文本帮助

运行

```
$ pydoc atexit
```

会在控制台上生成纯文本帮助，可能还会使用一个分页程序（如果已配置）。

16.1.2 HTML 帮助

pydoc 还会生成 HTML 输出，可能将一个静态文件写至一个本地目录，或者启动一个 Web 服务器在线浏览文档。

```
$ pydoc -w atexit
```

将在当前目录创建 atexit.html。

```
$ pydoc -p 5000
```

将启动一个 Web 服务器监听 <http://localhost:5000/>。这个服务器会在你浏览时动态生成文档。

16.1.3 交互式帮助

pydoc 还为 `__builtins__` 添加了一个函数 `help()`，从而可以从解释器提示窗口访问同样的信息。

```
$ python
```

```
Python 2.7 (r27:82508, Jul 3 2010, 21:12:11)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
```

```
>>> help('atexit')
Help on module atexit:
```

```
NAME
    atexit
```

```
...
```

参见：

pydoc (<http://docs.python.org/library/pydoc.html>) 这个模块的标准库文档。

inspect (18.4 节) 可以用 inspect 模块通过编程获取一个对象的 docstring。

16.2 doctest——通过文档完成测试

作用：编写自动化测试，作为模块文档的一部分。

Python 版本：2.1 及以后版本

doctest 会运行文档中嵌入的例子，并验证它们是否能生成所期望的结果，从而对源代码进行测试。其做法是解析帮助文档，找到例子，运行这些例子，然后将输出文本与所期望的值进行比较。很多开发人员发现 doctest 比 unittest 更易于使用，因为如果采用最简单的形式，使用 doctest 之前无须学习新的 API。不过，随着例子变得越来越复杂，由于缺乏固件管理，可能编写 doctest 测试比使用 unittest 更麻烦。

16.2.1 开始

建立 doctest 的第一步是使用交互式解释器创建例子，然后把这些例子复制粘贴到模块的 docstring 中。在这里，my_function() 给出两个例子。

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

运行这些测试时，要通过 -m 选项将 doctest 用作主程序。运行测试时通常不会生成输出，所以下面的例子包含了 -v 选项，以得到更详细的输出。

```
$ python -m doctest -v doctest_simple.py
```

```
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple
1 items passed all tests:
    2 tests in doctest_simple.my_function
```



```
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

例子并不总能独立作为一个函数的解释，所以 `doctest` 还允许有包围文本。它会查找以解释器提示符 (`>>>`) 开头的行，来找出测试用例的开始，用例以一个空行结束，或者以下一个解释器提示符结束。介于中间的文本会被忽略，它们可以有任何格式（只要看上去不像是一个测试用例）。

```
def my_function(a, b):
    """Returns a * b.

    Works with numbers:

    >>> my_function(2, 3)
    6

    and strings:

    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

更新的 docstring 中如果有包围文本，这些包围文本对人类读者来说更有用。由于它会被 `doctest` 忽略，所以结果是一样的。

```
$ python -m doctest -v doctest_simple_with_docs.py
```

```
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple_with_docs
1 items passed all tests:
    2 tests in doctest_simple_with_docs.my_function .
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

16.2.2 处理不可预测的输出

还有一些情况，可能无法预测准确的输出，不过仍可测试。例如，每次运行测试时本地日

期和时间值及对象 id 会改变，浮点值表示中使用的默认精度取决于编译器选项，另外对象的串表示可能不是确定性的。尽管这些条件可能不能控制，但是确实有一些技术可以处理。

例如，在 CPython 中，对象标识符基于保存对象的数据结构的内存地址。

```
class MyClass(object):
    pass

def unpredictable(obj):
    """Returns a new list containing obj.

    >>> unpredictable(MyClass())
    [<doctest_unpredictable.MyClass object at 0x10055a2d0>]
    """
    return [obj]
```

每次程序运行时，这些 id 值都会改变，因为这些值会加载到内存的不同部分。

```
$ python -m doctest -v doctest_unpredictable.py
```

```
Trying:
unpredictable(MyClass())
Expecting:
[<doctest_unpredictable.MyClass object at 0x10055a2d0>]
*****
File "doctest_unpredictable.py", line 16, in doctest_unpredictable.unpredictable
Failed example:
unpredictable(MyClass())
Expected:
[<doctest_unpredictable.MyClass object at 0x10055a2d0>]
Got:
[<doctest_unpredictable.MyClass object at 0x100ea3490>]
2 items had no tests:
doctest_unpredictable
doctest_unpredictable.MyClass
*****
1 items had failures:
1 of 1 in doctest_unpredictable.unpredictable
1 tests in 3 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

测试的值可能会以不可预测的方式改变时，如果具体值对于测试结果并不重要，可以使用 ELLIPSIS 选项来告诉 doctest 忽略验证值的某些部分。

```
class MyClass(object):
    pass

def unpredictable(obj):
```

```

    """Returns a new list containing obj.

    >>> unpredictable(MyClass()) #doctest: +ELLIPSIS
    [<doctest_ellipsis.MyClass object at 0x...>]
    """
    return [obj]

```

`unpredictable()` (#doctest: +ELLIPSIS) 调用后的注释告诉 doctest 打开这个测试的 ELLIPSIS 选项。... 将替换对象 id 中的内存地址，这样就会忽略期望值中的这一部分。实际输出将匹配，并通过测试。

```
$ python -m doctest -v doctest_ellipsis.py
```

```

Trying:
    unpredictable(MyClass()) #doctest: +ELLIPSIS
Expecting:
    [<doctest_ellipsis.MyClass object at 0x...>]
ok
2 items had no tests:
    doctest_ellipsis
    doctest_ellipsis.MyClass
1 items passed all tests:
    1 tests in doctest_ellipsis.unpredictable
1 tests in 3 items.
1 passed and 0 failed.
Test passed.

```

有些情况下，不能忽略不可预测的值，因为这会让测试不完备或不准确。例如，处理一些数据类型时，如果其数据串表示不一致，简单的测试很快会变得越来越复杂。举例来说，字典的串形式可能会根据增加键的顺序而改变。

```

keys = [ 'a', 'aa', 'aaa' ]

d1 = dict( (k,len(k)) for k in keys )
d2 = dict( (k,len(k)) for k in reversed(keys) )

print 'd1:', d1
print 'd2:', d2
print 'd1 == d2:', d1 == d2

s1 = set(keys)
s2 = set(reversed(keys))

print
print 's1:', s1
print 's2:', s2
print 's1 == s2:', s1 == s2

```



由于缓存冲突，两个字典的内部键列表顺序会有所不同，尽管它们包含相同的值，并被认为是相等的。集合（set）会使用相同的散列算法，并提供相同的行为。

```
$ python doctest_hashed_values.py
```

```
d1: {'a': 1, 'aa': 2, 'aaa': 3}
d2: {'aa': 2, 'a': 1, 'aaa': 3}
d1 == d2: True
s1: set(['a', 'aa', 'aaa'])
s2: set(['aa', 'a', 'aaa'])
s1 == s2: True
```

要处理这些潜在的差异，最好的方法是创建测试来生成不太可能改变的值。对于字典和集合，这可能意味着需要分别查找特定的键，并生成数据结构内容的一个有序列表，或者与一个字面值比较相等性而不是依赖于串表示。

```
def group_by_length(words):
    """Returns a dictionary grouping words into sets by length.

    >>> grouped = group_by_length(['python', 'module', 'of',
    ... 'the', 'week' ])
    >>> grouped == { 2:set(['of']),
    ...              3:set(['the']),
    ...              4:set(['week']),
    ...              6:set(['python', 'module']),
    ...              }
    True

    """
    d = {}
    for word in words:
        s = d.setdefault(len(word), set())
        s.add(word)
    return d
```

这个例子实际上会解释为两个单独的测试，第一个没有任何控制台输出，第二个会得到比较操作的布尔结果。

```
$ python -m doctest -v doctest_hashed_values_tests.py
```

```
Trying:
    grouped = group_by_length(['python', 'module', 'of',
    ... 'the', 'week' ])
Expecting nothing
ok
Trying:
    grouped == { 2:set(['of']),
                  3:set(['the']),
                  4:set(['week']),
```

```

        6:set(['python', 'module']),
    }

Expecting:
    True
ok
1 items had no tests:
    doctest_hashed_values_tests
1 items passed all tests:
    2 tests in doctest_hashed_values_tests.group_by_length
2 tests in 2 items.
2 passed and 0 failed.
Test passed.

```

16.2.3 Traceback

Traceback 是不断变化的数据的一个特殊情况。由于 traceback 中的路径取决于模块安装在给定系统文件系统上的具体位置，如果像其他输出一样处理，可能无法编写可移植的测试。

```

def this_raises():
    """This function always raises an exception.

    >>> this_raises()
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/no/such/path/doctest_tracebacks.py", line 14, in
        this_raises
          raise RuntimeError('here is the error')
    RuntimeError: here is the error
    """
    raise RuntimeError('here is the error')

```

doctest 做了一些特殊工作来识别 traceback，并忽略因系统不同可能改变的部分。

```
$ python -m doctest -v doctest_tracebacks.py
```

```

Trying:
    this_raises()
Expecting:
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/no/such/path/doctest_tracebacks.py", line 14, in
        this_raises
          raise RuntimeError('here is the error')
    RuntimeError: here is the error
ok
1 items had no tests:
    doctest_tracebacks
1 items passed all tests:
    1 tests in doctest_tracebacks.this_raises

```




```
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

实际上，整个 traceback 体都将被忽略，可以略去。

```
def this_raises():
    """This function always raises an exception.

    >>> this_raises()
    Traceback (most recent call last):
    RuntimeError: here is the error
    """
    raise RuntimeError('here is the error')
```

doctest 看到一个 traceback 首部行时（可能是“Traceback (most recent call last):”或“Traceback (innermost last):”，取决于所用的 Python 版本），它会跳过，继续查找异常类型和消息，完全忽略中间的各行。

```
$ python -m doctest -v doctest_tracebacks_no_body.py
```

```
Trying:
    this_raises()
Expecting:
    Traceback (most recent call last):
    RuntimeError: here is the error
ok
1 items had no tests:
    doctest_tracebacks_no_body
1 items passed all tests:
    1 tests in doctest_tracebacks_no_body.this_raises
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

16.2.4 避开空白符

在实际应用中，输出通常包括空白符，如空行、制表符和多余的空格，目的是使输出更可读。尤其是空行会导致 doctest 出现问题，因为一般会用空行作为测试的分界线。

```
def double_space(lines):
    """Prints a list of lines double-spaced.

    >>> double_space(['Line one.', 'Line two.'])
    Line one.

    Line two.

    """
```

```

    for l in lines:
        print l
        print
    return

```

`double_space()` 取一个输入行列表，在输入行之间加入空行，从而用双倍间隔打印。

```
$ python -m doctest doctest_blankline_fail.py
```

```

*****
File "doctest_blankline_fail.py", line 13, in doctest_blankline
_fail.double_space
Failed example:
double_space(['Line one.', 'Line two.'])
Expected:
Line one.
Got:
Line one.
<BLANKLINE>
Line two.
<BLANKLINE>
*****
1 items had failures:
1 of 1 in doctest_blankline_fail.double_space
***Test Failed*** 1 failures.

```

这个测试会失败，因为它把 docstring 中包含 Line one. 那一行后面的空行解释为示例输出的末尾。为了与空行匹配，要把示例输入中的空行替换为 `<BLANKLINE>`。

```

def double_space(lines):
    """Prints a list of lines double-spaced.

    >>> double_space(['Line one.', 'Line two.'])
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
    """
    for l in lines:
        print l
        print
    return

```

完成比较之前，doctest 将具体的空行替换为相同的字面量，所以现在具体值和期望值匹配，测试通过。

```
$ python -m doctest -v doctest_blankline.py
```

```

Trying:
    double_space(['Line one.', 'Line two.'])

```

```

Expecting:
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
ok
1 items had no tests:
    doctest_blankline
1 items passed all tests:
    1 tests in doctest_blankline.double_space
1 tests in 2 items.
1 passed and 0 failed.
Test passed.

```

使用文本比较完成测试还存在一个麻烦：嵌入的空白符还可能导致测试出现棘手的问题。下面这个例子在 6 后面有一个额外的空格。

```

def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b

```

这个额外的空格可能是因为复制粘贴错误而引入代码，不过由于它们位于行末尾，可能在源文件中不被注意，在测试失败报告中也不可见。

```
$ python -m doctest -v doctest_extra_space.py
```

```

Trying:
my_function(2, 3)
Expecting:
6
*****
File "doctest_extra_space.py", line 12, in doctest_extra_space.
my_function
Failed example:
my_function(2, 3)
Expected:
6
Got:
6
Trying:
my_function('a', 3)
Expecting:
'aaa'
ok

```

```

1 items had no tests:
doctest_extra_space
*****
1 items had failures:
1 of 2 in doctest_extra_space.my_function
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.

```

使用某个基于 diff 的报告选项，如 REPORT_NDIFF，可以更详细地显示实际值和期望值之间的差异，这样就会看到这个额外的空格。

```

def my_function(a, b):
    """
    >>> my_function(2, 3) #doctest: +REPORT_NDIFF
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b

```

对于有些输出，也可以使用统一 diff (REPORT_UDIFF) 和上下文 diff (REPORT_CDIF)，这些格式将更可读。

```
$ python -m doctest -v doctest_ndiff.py
```

Trying:

```
my_function(2, 3) #doctest: +REPORT_NDIFF
```

Expecting:

```
6
```

```

*****
File "doctest_ndiff.py", line 12, in doctest_ndiff.my_function
Failed example:

```

```
my_function(2, 3) #doctest: +REPORT_NDIFF
```

Differences (ndiff with -expected +actual):

```

- 6
? -
+ 6

```

Trying:

```
my_function('a', 3)
```

Expecting:

```
'aaa'
```

ok

```
1 items had no tests:
```

```
doctest_ndiff
```

```
*****
```

```
1 items had failures:
```

```
1 of 2 in doctest_ndiff.my_function
```

```
2 tests in 2 items.
```



```
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

有些情况下，可能要在测试的示例输出中添加额外的空白符，而让 doctest 忽略这些空白符。例如，尽管有些数据结构的表示可以显示在一行上，不过多行显示可能更易读。

```
def my_function(a, b):
    """Returns a * b.

    >>> my_function(['A', 'B'], 3) #doctest: +NORMALIZE_WHITESPACE
    ['A', 'B',
     'A', 'B',
     'A', 'B',]

    This does not match because of the extra space after the [ in
    the list.

    >>> my_function(['A', 'B'], 2) #doctest: +NORMALIZE_WHITESPACE
    [ 'A', 'B',
      'A', 'B', ]
    """
    return a * b
```

NORMALIZE_WHITESPACE 打开时，实际值和期望值中的空白符会认为是匹配的。如果输出中不存在空白符，期望值中就不能增加空白符，不过空白符序列和实际的空白字符不需要一致。第一个测试示例满足这个原则，测试通过，尽管这里有额外的空格和换行。第二个测试示例在 [“and before”] 后面有额外的空白符，所以失败。

```
$ python -m doctest -v doctest_normalize_whitespace.py
```

```
Trying:
my_function(['A', 'B'], 3) #doctest: +NORMALIZE_WHITESPACE
Expecting:
['A', 'B',
 'A', 'B',
 'A', 'B',]
*****
File "doctest_normalize_whitespace.py", line 13, in doctest_normalize_whitespace.my_function
Failed example:
my_function(['A', 'B'], 3) #doctest: +NORMALIZE_WHITESPACE
Expected:
['A', 'B',
 'A', 'B',
 'A', 'B',]
Got:
['A', 'B', 'A', 'B', 'A', 'B']
Trying:
```

```
my_function(['A', 'B'], 2) #doctest: +NORMALIZE_WHITESPACE
Expecting:
[ 'A', 'B',
  'A', 'B', ]
*****
File "doctest_normalize_whitespace.py", line 21, in doctest_normalize_whitespace.my_function
Failed example:
my_function(['A', 'B'], 2) #doctest: +NORMALIZE_WHITESPACE
Expected:
[ 'A', 'B',
  'A', 'B', ]
Got:
['A', 'B', 'A', 'B']
1 items had no tests:
doctest_normalize_whitespace
*****
1 items had failures:
2 of 2 in doctest_normalize_whitespace.my_function
2 tests in 2 items.
0 passed and 2 failed.
***Test Failed*** 2 failures.
```

16.2.5 测试位置

到目前为止的例子中，所有测试都写在所测试的函数的 docstring 中。对于查看 docstring 的用户来说这很方便，可以帮助他们使用这个函数（特别是利用 pydoc），不过 doctest 还会在其他地方查找测试。一个很明显的地方就是模块中的其他 docstring。

```
#!/usr/bin/env python
# encoding: utf-8

"""Tests can appear in any docstring within the module.

Module-level tests cross class and function boundaries.

>>> A('a') == B('b')
False
"""

class A(object):
    """Simple class.

    >>> A('instance_name').name
    'instance_name'
    """
    def __init__(self, name):
```

```
        self.name = name
    def method(self):
        """Returns an unusual value.

    >>> A('name').method()
    'eman'
    """
    return ''.join(reversed(list(self.name)))

class B(A):
    """Another simple class.

    >>> B('different_name').name
    'different_name'
    """
```

模块级、类级和函数级的 docstring 都可以包含测试。

```
$ python -m doctest -v doctest_docstrings.py
```

```
Trying:
    A('a') == B('b')
Expecting:
    False
ok
Trying:
    A('instance_name').name
Expecting:
    'instance_name'
ok
Trying:
    A('name').method()
Expecting:
    'eman'
ok
Trying:
    B('different_name').name
Expecting:
    'different_name'
ok
1 items had no tests:
    doctest_docstrings.A.__init__
4 items passed all tests:
    1 tests in doctest_docstrings
    1 tests in doctest_docstrings.A
    1 tests in doctest_docstrings.A.method
    1 tests in doctest_docstrings.B
4 tests in 5 items.
4 passed and 0 failed.
```



Test passed.

有些情况下，模块的测试应当包含在源代码中，而不是放在模块的帮助文本中，所以需要把测试放在 docstring 以外的位置。doctest 还会查找一个模块级变量，名为 `__test__`，用它来找到其他测试。`__test__` 的值应当是一个字典，将测试集合名（字符串）映射到字符串、模块、类或函数。

```
import doctest_private_tests_external

__test__ = {
    'numbers': """
>>> my_function(2, 3)
6

>>> my_function(2.0, 3)
6.0
""",

    'strings': """
>>> my_function('a', 3)
'aaa'

>>> my_function(3, 'a')
'aaa'
""",

    'external': doctest_private_tests_external,
}

def my_function(a, b):
    """Returns a * b
    """
    return a * b
```

如果与一个键关联的值是字符串，会处理为一个 docstring，并在其中扫描测试。如果值是一个类或函数，doctest 会递归地搜索 docstring，然后在其中扫描测试。在这个例子中，模块 `doctest_private_tests_external` 的 docstring 中有一个测试。

```
#!/usr/bin/env python
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann. All rights reserved.
#
"""External tests associated with doctest_private_tests.py.

>>> my_function(['A', 'B', 'C'], 2)
['A', 'B', 'C', 'A', 'B', 'C']
"""
```


扫描示例文件之后，doctest 总共找到 5 个要运行的测试。

```
$ python -m doctest -v doctest_private_tests.py
```

```
Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(3, 'a')
Expecting:
    'aaa'
ok
2 items had no tests:
    doctest_private_tests
    doctest_private_tests.my_function
3 items passed all tests:
    1 tests in doctest_private_tests.__test__.external
    2 tests in doctest_private_tests.__test__.numbers
    2 tests in doctest_private_tests.__test__.strings
5 tests in 5 items.
5 passed and 0 failed.
Test passed.
```

16.2.6 外部文档

将测试混合在常规代码中并不是使用 doctest 的惟一办法。也可以使用外部工程文档文件（如 reStructuredText 文件）中嵌入的示例。

```
def my_function(a, b):
    """Returns a*b
    """
    return a * b
```

这个示例模块的帮助保存在一个单独的文件 `doctest_in_help.rst` 中。这些例子展示了帮助文本中包含有如何使用模块的说明，可以使用 `doctest` 查找并运行。

```
=====
How to Use doctest_in_help.py
=====
```

```
This library is very simple, since it only has one function called
`my_function()`.
```

```
Numbers
=====
```

```
`my_function()` returns the product of its arguments. For numbers,
that value is equivalent to using the `*` operator.
```

```
::
```

```
>>> from doctest_in_help import my_function
>>> my_function(2, 3)
6
```

It also works with floating-point values.

```
::
```

```
>>> my_function(2.0, 3)
6.0
```

```
Non-Numbers
=====
```

```
Because `*` is also defined on data types other than numbers,
`my_function()` works just as well if one of the arguments is a
string, a list, or a tuple.
```

```
::
```

```
>>> my_function('a', 3)
'aaa'

>>> my_function(['A', 'B', 'C'], 2)
['A', 'B', 'C', 'A', 'B', 'C']
```

文本文件中的测试可以从命令行运行，这与 Python 源模块类似。

```
$ python -m doctest -v doctest_in_help.rst
```

Trying:

```

    from doctest_in_help import my_function
Expecting nothing
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
1 items passed all tests:
   5 tests in doctest_in_help.rst
5 tests in 1 items.
5 passed and 0 failed.
Test passed.

```

正常情况下，doctest 会建立测试执行环境来包含所测试模块的成员，这样测试就不需要再显式地导入这个模块。不过，在这种情况下，测试不在 Python 模块中定义，doctest 不知道如何建立全局命名空间，所以这些例子需要自行完成导入工作。一个给定文件中的所有测试都共享相同的执行上下文，所以在文件最前面导入一次模块就足够了。

16.2.7 运行测试

前面的例子都使用 doctest 内置的命令行测试运行工具。对于单个模块来说，这很容易也很方便，不过随着包划分为多个文件，这很快会变得很麻烦。还有很多其他方法来运行测试。

由模块运行

可以在模块最下面包含对源代码运行 doctest 的指令。

```

def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)

```

```

    'aaa'
    """
    return a * b

if __name__ == '__main__':
    import doctest
    doctest.testmod()

```

只有当前模块名是 `__main__` 时才会调用 `testmod()`，这可以确保仅当模块作为主程序调用时才运行测试。

```
$ python doctest_testmod.py -v
```

```

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    __main__
1 items passed all tests:
    2 tests in __main__.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.

```

`testmod()` 的第一个参数是一个模块，包含需要扫描的代码（检查其中是否有测试）。其他测试脚本可以使用这个特性导入实际代码，并依次运行各个模块中的测试。

```

import doctest_simple

if __name__ == '__main__':
    import doctest
    doctest.testmod(doctest_simple)

```

通过导入各个模块并运行其测试，可以为工程构造一个测试套件。

```
$ python doctest_testmod_other_module.py -v
```

```

Trying:
    my_function(2, 3)
Expecting:
    6
ok

```



```

Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple
1 items passed all tests:
    2 tests in doctest_simple.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.

```

由文件运行

`testfile()` 的做法类似于 `testmod()`，允许在测试程序中从一个外部文件显式调用测试。

```

import doctest

if __name__ == '__main__':
    doctest.testfile('doctest_in_help.rst')

```

`testmod()` 和 `testfile()` 包括一些可选的参数，能通过 `doctest` 选项来控制测试的行为。关于这些特性的更多详细信息可以参考标准库文档。大多数情况下并不需要这些特性。

```

$ python doctest_testfile.py -v

Trying:
    from doctest_in_help import my_function
Expecting nothing
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:

```



```

    ['A', 'B', 'C', 'A', 'B', 'C']
ok
1 items passed all tests:
  5 tests in doctest_in_help.rst
5 tests in 1 items.
5 passed and 0 failed.
Test passed.

```

Unittest 套件

unittest 和 doctest 同时用于在不同情况测试相同代码时，可以使用 doctest 中的 unittest 集成让测试一起运行。有两个类（DocTestSuite 和 DocFileSuite）可以创建与 unittest 测试运行工具 API 兼容的测试套件。

```

import doctest
import unittest

import doctest_simple

suite = unittest.TestSuite()
suite.addTest(doctest.DocTestSuite(doctest_simple))
suite.addTest(doctest.DocFileSuite('doctest_in_help.rst'))

runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)

```

各个源文件的测试会合并到一个结果，而不是分别报告。

```
$ python doctest_unittest.py
```

```

my_function (doctest_simple)
Doctest: doctest_simple.my_function ... ok
doctest_in_help.rst
Doctest: doctest_in_help.rst ... ok

```

```
-----
Ran 2 tests in 0.006s
```

```
OK
```

16.2.8 测试上下文

doctest 运行测试时创建的执行上下文包含测试模块的模块级全局变量的一个副本。每个测试源（函数、类、模块）都有自己的一组全局值，使测试在某种程度上相互隔离，从而不太可能相互干扰。

```

class TestGlobals(object):

    def one(self):
        """
        >>> var = 'value'

```

```

>>> 'var' in globals()
True
"""

def two(self):
    """
    >>> 'var' in globals()
    False
    """

```

TestGlobals 有两个方法：one() 和 two()。one() 的 docstring 中的测试设置了一个全局变量，two() 的测试要查找这个变量（期望找不到这个变量）。

```
$ python -m doctest -v doctest_test_globals.py
```

```

Trying:
    var = 'value'
Expecting nothing
ok
Trying:
    'var' in globals()
Expecting:
    True
ok
Trying:
    'var' in globals()
Expecting:
    False
ok
2 items had no tests:
    doctest_test_globals
    doctest_test_globals.TestGlobals
2 items passed all tests:
    2 tests in doctest_test_globals.TestGlobals.one
    1 tests in doctest_test_globals.TestGlobals.two
3 tests in 4 items.
3 passed and 0 failed.
Test passed.

```

不过，这并不表示测试不能相互干扰，如果测试要改变模块中定义的可变变量的内容，反而希望它们能交互。

```

_module_data = {}

class TestGlobals(object):

    def one(self):
        """
        >>> TestGlobals().one()

```

```

>>> 'var' in _module_data
True
"""
_module_data['var'] = 'value'

def two(self):
    """
    >>> 'var' in _module_data
    False
    """

```

one() 的测试改变了模块变量 `_module_data`，导致 `two()` 的测试失败。

```
$ python -m doctest -v doctest_mutable_globals.py
```

```

Trying:
TestGlobals().one()
Expecting nothing
ok
Trying:
'var' in _module_data
Expecting:
True
ok
Trying:
'var' in _module_data
Expecting:
False
*****
File "doctest_mutable_globals.py", line 24, in doctest_mutable_
globals.TestGlobals.two
Failed example:
'var' in _module_data
Expected:
False
Got:
True
2 items had no tests:
doctest_mutable_globals
doctest_mutable_globals.TestGlobals
1 items passed all tests:
2 tests in doctest_mutable_globals.TestGlobals.one
*****
1 items had failures:
1 of 1 in doctest_mutable_globals.TestGlobals.two
3 tests in 4 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.

```


如果测试需要全局值，例如对应一个环境进行参数化，可以将值传递到 `testmod()` 和 `testfile()`，使用调用者控制的数据建立上下文。

参见：

`doctest` (<http://docs.python.org/library/doctest.html>) 这个模块的标准库文档。

The Mighty Dictionary (<http://blip.tv/file/3332763>) Brandon Rhodes 在 PyCon 2010 关于 dict 内部操作的演示稿。

`difflib` (1.4 节) Python 的顺序差异计算库，用于生成 `ndiff` 输出。

`Sphinx` (<http://sphinx.pocoo.org/>) 除了作为 Python 标准库的文档处理工具外，`Sphinx` 已经被很多第三方项目所采用，因为它不仅易于使用，还可以采用多种数字和打印格式生成简洁的输出。`Sphinx` 包括一个扩展包，可以在它处理文档源文件时运行 `doctest`，以确保例子正确。

`nose` (<http://somethingaboutorange.com/mrl/projects/nose/>) 提供 `doctest` 支持的第三方测试运行工具。

`py.test` (<http://codespeak.net/py/dist/test/>) 提供 `doctest` 支持的第三方测试运行工具。

`Manuel` (<http://packages.python.org/manuel/>) 基于文档的第三方测试运行工具，提供更高级的测试用例抽取以及与 `Sphinx` 的集成。

16.3 unittest——自动测试框架

作用：自动测试框架。

Python 版本：2.1 及以后版本

Python 的 `unittest` 模块（有时称为 `PyUnit`）基于 Kent Beck 和 Erich Gamma 提出的 XUnit 框架设计。同样的模式在很多其他语言中都有出现，包括 C、Perl、Java 和 Smalltalk。`unittest` 实现的框架支持固件和测试套件，还提供一个测试运行工具来完成自动测试。

16.3.1 基本测试结构

按照 `unittest` 的定义，测试有两个部分：管理测试依赖库的代码（称为“固件”）和测试本身。单个测试通过派生 `Test-Case` 并覆盖或添加适当的方法来创建。例如：

```
import unittest

class SimplisticTest(unittest.TestCase):

    def test(self):
        self.failUnless(True)

if __name__ == '__main__':
    unittest.main()
```

在这个例子中，`SimplisticTest` 有一个 `test()` 方法，如果 `True` 总为 `False` 则该方法失败。

16.3.2 运行测试

运行 unittest 测试时，最容易的方法是在每个测试文件最后包括以下代码：

```
if __name__ == '__main__':
    unittest.main()
```

然后只需从命令行直接运行脚本。

```
$ python unittest_simple.py
```

```
.
-----
Ran 1 test in 0.000s

OK
```

这里的简略输出包括测试花费的时间，并为每个测试提供了一个状态指示符（输出第一行上的“.”表示一个测试通过）。要得到更详细的测试结果，可以包括 `-v` 选项：

```
$ python unittest_simple.py -v
```

```
test (__main__.SimplisticTest) ... ok
```

```
-----
Ran 1 test in 0.000s

OK
```

16.3.3 测试结果

测试有 3 种可能的结果，如表 16.1 所述。

没有明确的方法让一个测试“通过”，所以一个测试的状态取决于是否出现异常。

表 16.1 测试用例结果

结 果	描 述
ok	测试通过
FAIL	测试没有通过，产生一个 <code>AssertionError</code> 异常
ERROR	测试产生 <code>AssertionError</code> 以外的某个异常

```
import unittest

class OutcomeTest(unittest.TestCase):
    def testPass(self):
        return

    def testFail(self):
```

```

        self.failIf(True)

    def testError(self):
        raise RuntimeError('Test error!')

if __name__ == '__main__':
    unittest.main()

```

一个测试失败或生成一个错误时，输出中会包含 traceback。

```
$ python unittest_outcomes.py
```

```
EF.
```

```
=====
ERROR: testError (__main__.OutcomesTest)
-----
```

```
Traceback (most recent call last):
```

```
  File "unittest_outcomes.py", line 42, in testError
    raise RuntimeError('Test error!')
```

```
RuntimeError: Test error!
```

```
=====
FAIL: testFail (__main__.OutcomesTest)
-----
```

```
Traceback (most recent call last):
```

```
  File "unittest_outcomes.py", line 39, in testFail
    self.failIf(True)
```

```
AssertionError: True is not False
```

```
-----
Ran 3 tests in 0.001s
```

```
FAILED (failures=1, errors=1)
```

在前面的例子中，testFail() 失败，traceback 显示了失败代码所在的那一行。不过，要由读测试输出的人来查看代码，明确失败测试的含义。

```
import unittest
```

```
class FailureMessageTest(unittest.TestCase):
```

```
    def testFail(self):
        self.failIf(True, 'failure message goes here')
```

```
if __name__ == '__main__':
    unittest.main()
```

为了更容易地理解一个测试失败的实质，fail*() 和 assert*() 方法都接受一个参数 msg，可以用来生成一个更详细的错误消息。

```
$ python unittest_failwithmessage.py -v

testFail (__main__.FailureMessageTest) ... FAIL

=====
FAIL: testFail (__main__.FailureMessageTest)
-----
Traceback (most recent call last):
  File "unittest_failwithmessage.py", line 36, in testFail
    self.failIf(True, 'failure message goes here')
AssertionError: failure message goes here
-----

Ran 1 test in 0.000s

FAILED (failures=1)
```

16.3.4 断言真值

大多数测试会断言某个条件的真值。编写真值检查测试有几种不同的方法，取决于测试作者的偏好以及所测试的代码的预期结果。

```
import unittest

class TruthTest(unittest.TestCase):

    def testFailUnless(self):
        self.failUnless(True)
    def testAssertTrue(self):
        self.assertTrue(True)

    def testFailIf(self):
        self.failIf(False)

    def testAssertFalse(self):
        self.assertFalse(False)

if __name__ == '__main__':
    unittest.main()
```

如果代码生成一个可能为 `true` 的值，就应当使用方法 `failUnless()` 和 `assertTrue()`。如果代码生成一个 `false` 值，方法 `failIf()` 和 `assertFalse()` 则更有意义。

```
$ python unittest_truth.py -v

testAssertFalse (__main__.TruthTest) ... ok
testAssertTrue (__main__.TruthTest) ... ok
testFailIf (__main__.TruthTest) ... ok
testFailUnless (__main__.TruthTest) ... ok
```

```
-----
Ran 4 tests in 0.000s
```

```
OK
```

16.3.5 测试相等性

作为一种特殊情况，unittest 还包括测试两个值相等性的方法。

```
import unittest

class EqualityTest(unittest.TestCase):

    def testExpectEqual(self):
        self.failUnlessEqual(1, 3-2)

    def testExpectEqualFails(self):
        self.failUnlessEqual(2, 3-2)
    def testExpectNotEqual(self):
        self.failIfEqual(2, 3-2)

    def testExpectNotEqualFails(self):
        self.failIfEqual(1, 3-2)

if __name__ == '__main__':
    unittest.main()
```

如果失败，这些特殊的测试方法会生成错误消息，其中包括所比较的值。

```
$ python unittest_equality.py -v
```

```
testExpectEqual (__main__.EqualityTest) ... ok
testExpectEqualFails (__main__.EqualityTest) ... FAIL
testExpectNotEqual (__main__.EqualityTest) ... ok
testExpectNotEqualFails (__main__.EqualityTest) ... FAIL

=====
FAIL: testExpectEqualFails (__main__.EqualityTest)
=====
Traceback (most recent call last):
  File "unittest_equality.py", line 39, in testExpectEqualFails
    self.failUnlessEqual(2, 3-2)
AssertionError: 2 != 1

=====
FAIL: testExpectNotEqualFails (__main__.EqualityTest)
=====
Traceback (most recent call last):
```

```
File "unittest_equality.py", line 45, in testExpectNotEqualFails
    self.failIfEqual(1, 3-2)
AssertionError: 1 == 1
```

```
-----
Ran 4 tests in 0.001s
```

```
FAILED (failures=2)
```

16.3.6 近似相等

除了严格相等性外，还可以使用 `failIfAlmostEqual()` 和 `failUnlessAlmostEqual()` 测试浮点数的近似相等性。

```
import unittest

class AlmostEqualTest(unittest.TestCase):

    def testEqual(self):
        self.failUnlessEqual(1.1, 3.3-2.2)

    def testAlmostEqual(self):
        self.failUnlessAlmostEqual(1.1, 3.3-2.2, places=1)

    def testNotAlmostEqual(self):
        self.failIfAlmostEqual(1.1, 3.3-2.0, places=1)

if __name__ == '__main__':
    unittest.main()
```

参数是要比较的值以及测试所用的小数位。

```
$ python unittest_almostequal.py
```

```
.F.
```

```
=====
FAIL: testEqual (__main__.AlmostEqualTest)
```

```
-----
Traceback (most recent call last):
```

```
File "unittest_almostequal.py", line 36, in testEqual
    self.failUnlessEqual(1.1, 3.3-2.2)
AssertionError: 1.1 != 1.0999999999999996
```

```
-----
Ran 3 tests in 0.001s
```

```
FAILED (failures=1)
```

16.3.7 测试异常

正如前面提到的，如果测试产生 `AssertionError` 以外的一个异常，会被处理为一个错误。这对于发现错误很有用，还有助于修改现有测试覆盖的代码。不过，有些情况下，测试要验证代码确实会产生一个异常。为对象的属性提供一个非法值就是这样一个例子。在这些情况下，与在测试中捕获异常相比，`failUnlessRaises()` 或 `assertRaises()` 可以使代码更简洁。比较以下两个测试。

```
import unittest

def raises_error(*args, **kwargs):
    raise ValueError('Invalid value: ' + str(args) + str(kwargs))

class ExceptionTest(unittest.TestCase):

    def testTrapLocally(self):
        try:
            raises_error('a', b='c')
        except ValueError:
            pass
        else:
            self.fail('Did not see ValueError')

    def testFailUnlessRaises(self):
        self.failUnlessRaises(ValueError, raises_error, 'a', b='c')

if __name__ == '__main__':
    unittest.main()
```

这两个测试的结果是相同的，不过第二个使用 `failUnlessRaises()` 的测试更为简洁。

```
$ python unittest_exception.py -v
```

```
testFailUnlessRaises (__main__.ExceptionTest) ... ok
testTrapLocally (__main__.ExceptionTest) ... ok
```

```
-----
Ran 2 tests in 0.000s
```

```
OK
```

16.3.8 测试固件

固件是测试所需的外部资源。例如，一个类的所有测试可能都需要另一个类的实例（用来提供配置设置）或另一个共享资源。其他测试固件包括数据库连接和临时文件（很多人可能对此有争议，认为使用外部资源会使这些测试不再是“单元”测试，但它们仍是测试，而且仍然很有用）。`TestCase` 包括一个特殊的 `hook`，用来配置和清理测试所需的所有固件。配置固件需

要覆盖 setUp()。要完成清理，则要覆盖 tearDown()。

```
import unittest

class FixturesTest(unittest.TestCase):

    def setUp(self):
        print 'In setUp()'
        self.fixture = range(1, 10)

    def tearDown(self):
        print 'In tearDown()'
        del self.fixture

    def test(self):
        print 'In test()'
        self.failUnlessEqual(self.fixture, range(1, 10))

if __name__ == '__main__':
    unittest.main()
```

运行这个示例测试时，固件和测试方法执行的顺序显而易见。

```
$ python -u unittest_fixtures.py
```

```
In setUp()
In test()
In tearDown()
```

```
-----
Ran 1 test in 0.000s
```

```
OK
```

16.3.9 测试套件

标准库文档描述了如何手动地组织测试套件。对于很大的代码基，其中相关的测试并不都在同一个地方，对于这种情况，自动测试发现则更可管理。利用 nose 和 py.test 等工具，当测试分布在多个文件和目录中时，管理测试会更为容易。

参见：

unittest (<http://docs.python.org/lib/module-unittest.html>) 这个模块的标准库文档。

doctest (16.2 节) 运行嵌入在 docstring 或外部文档文件中的测试的另一种候选方式。

nose (<http://somethingaboutorange.com/mrl/projects/nose/>) 一个更复杂的测试管理器。

py.test (<http://codespeak.net/py/dist/test/>) 一个第三方测试运行工具。

unittest2 (<http://pypi.python.org/pypi/unittest2>) 正在进行的 unittest 改进。

16.4 traceback——异常和栈轨迹

作用：抽取、格式化和打印异常及栈轨迹。

Python 版本：1.4 及以后版本

traceback 模块处理调用栈来生成错误消息。traceback 是从异常处理程序沿调用链向下直到产生异常那一点的栈轨迹。也可以在当前调用栈从调用位置（没有错误上下文）向上访问 traceback，这对于查找进入函数的路径很有用。

traceback 中的函数通常可以分为几类。有些函数用于从当前运行时环境（可能是一个 traceback 的异常处理程序或者是常规的栈）抽取原始 traceback。所抽取的栈轨迹是一个元组序列，元组中包含文件名、行号、函数名和源代码行文本。

一旦抽取了栈轨迹，可以使用类似 `format_exception()`、`format_stack()` 等等函数进行格式化。格式化函数会返回一个字符串列表，其中对消息进行了格式化以便打印。还有一些用于打印格式化值的简写函数。

traceback 中的函数默认地会模拟交互式解释器的行为，不仅如此，它们对于另外一些情况下异常的处理也很有用，这些情况下并不需要完整的栈轨迹转储到控制台。例如，一个 Web 应用可能需要格式化 traceback，从而采用 HTML 格式很好地显示，或者一个 IDE 可以将栈轨迹的元素转换为一个可点击的列表，以使用户浏览源文本。

16.4.1 支持函数

本节中的例子使用了模块 `traceback_example.py`。

```
import traceback
import sys
def produce_exception(recursion_level=2):
    sys.stdout.flush()
    if recursion_level:
        produce_exception(recursion_level-1)
    else:
        raise RuntimeError()

def call_function(f, recursion_level=2):
    if recursion_level:
        return call_function(f, recursion_level-1)
    else:
        return f()
```

16.4.2 处理异常

要处理异常报告，最简单的方法是利用 `print_exc()`。它使用 `sys.exc_info()` 来得到当前线程的异常信息，格式化结果，并把文本打印到一个文件句柄（默认为 `sys.stderr`）。

```
import traceback
import sys
```

```

from traceback_example import produce_exception...

print 'print_exc() with no exception:'
traceback.print_exc(file=sys.stdout)
print

try:
    produce_exception()
except Exception, err:
    print 'print_exc():'
    traceback.print_exc(file=sys.stdout)
    print
    print 'print_exc(1):'
    traceback.print_exc(limit=1, file=sys.stdout)

```

在这个例子中，替换为 `sys.stdout` 的文件句柄，使信息消息和 `traceback` 消息能够正确地混合。

```
$ python traceback_print_exc.py
```

```

print_exc() with no exception:
None
print_exc():
Traceback (most recent call last):
  File "traceback_print_exc.py", line 20, in <module>
    produce_exception()
  File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 16, in produce_exception
    produce_exception(recursion_level-1)
  File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 16, in produce_exception
    produce_exception(recursion_level-1)
  File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 18, in produce_exception
    raise RuntimeError()
RuntimeError

print_exc(1):
Traceback (most recent call last):
  File "traceback_print_exc.py", line 20, in <module>
    produce_exception()
RuntimeError

```

`print_exc()` 就是 `print_exception()` 的一个简写形式，它需要显式参数。

```

import traceback
import sys

from traceback_example import produce_exception

```

```

try:
    produce_exception()
except Exception, err:
    print 'print_exception():'
    exc_type, exc_value, exc_tb = sys.exc_info()
    traceback.print_exception(exc_type, exc_value, exc_tb)

```

print_exception() 的参数由 sys.exc_info() 生成。

```
$ python traceback_print_exception.py
```

Traceback (most recent call last):

```

File "traceback_print_exception.py", line 16, in <module>
    produce_exception()
File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 16, in produce_exception
    produce_exception(recursion_level-1)
File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 16, in produce_exception
    produce_exception(recursion_level-1)
File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 18, in produce_exception
    raise RuntimeError()

```

RuntimeError

```
print_exception():
```

print_exception() 使用 format_exception() 来准备文本。

```

import traceback
import sys
from pprint import pprint

from traceback_example import produce_exception

```

```

try:
    produce_exception()
except Exception, err:
    print 'format_exception():'
    exc_type, exc_value, exc_tb = sys.exc_info()
    pprint(traceback.format_exception(exc_type, exc_value, exc_tb))

```

format_exception() 使用同样的 3 个参数：异常类型、异常值和 traceback。

```
$ python traceback_format_exception.py
```

```

format_exception():
['Traceback (most recent call last):\n',
 '  File "traceback_format_exception.py", line 17, in <module>\n',
 '    produce_exception()\n',
 '  File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 16, in produce_exception\n',
 '    produce_exception(recursion_level-1)\n',
 '  File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 16, in produce_exception\n',
 '    produce_exception(recursion_level-1)\n',
 '  File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 18, in produce_exception\n',
 '    raise RuntimeError()\n',
 'RuntimeError\n']

```

```

ption(recursion_level-1)\n',
' File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/tr
aceback_example.py", line 16, in produce_exception\n    produce_exce
ption(recursion_level-1)\n',
' File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/tr
aceback_example.py", line 18, in produce_exception\n    raise Runtim
eError()\n',
'RuntimeError\n']

```

要以另外某种方式处理 `traceback`，如以不同方式格式化，可以使用 `extract_tb()` 得到采用一种有用格式的数据。

```

import traceback
import sys
import os
from traceback_example import produce_exception

try:
    produce_exception()
except Exception, err:
    print 'format_exception():'
    exc_type, exc_value, exc_tb = sys.exc_info()
    for tb_info in traceback.extract_tb(exc_tb):
        filename, linenum, funcname, source = tb_info
        print '%-23s:%s "%s" in %s()' % \
            (os.path.basename(filename),
             linenum,
             source,
             funcname)

```

返回值是一个项列表，这些项来自 `traceback` 表示的栈中的各层。每一项是一个元组，包括 4 个部分：源文件名、该文件中的行号、函数名和该行的源文本（如果可以得到源文本，会去除其中的空白符）。

```
$ python traceback_extract_tb.py
```

```

format_exception():
traceback_extract_tb.py:16 "produce_exception()" in <module>()
traceback_example.py    :16 "produce_exception(recursion_level-1)" in
    produce_exception()
traceback_example.py    :16 "produce_exception(recursion_level-1)" in
    produce_exception()
traceback_example.py    :18 "raise RuntimeError()" in produce_excepti
on()

```

16.4.3 处理栈

还有一组类似的函数，可以对当前调用栈而不是 `traceback` 完成同样的操作，`print_stack()`

会打印当前栈，而不生成异常。

```
import traceback
import sys

from traceback_example import call_function

def f():
    traceback.print_stack(file=sys.stdout)

print 'Calling f() directly:'
f()

print
print 'Calling f() from 3 levels deep:'
call_function(f)
```

输出看起来就像是一个 `traceback` 但没有错误消息。

```
$ python traceback_print_stack.py
```

Calling f() directly:

```
File "traceback_print_stack.py", line 19, in <module>
    f()
File "traceback_print_stack.py", line 16, in f
    traceback.print_stack(file=sys.stdout)
```

Calling f() from 3 levels deep:

```
File "traceback_print_stack.py", line 23, in <module>
    call_function(f)
File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 22, in call_function
    return call_function(f, recursion_level-1)
File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 22, in call_function
    return call_function(f, recursion_level-1)
File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/traceback_example.py", line 24, in call_function
    return f()
File "traceback_print_stack.py", line 16, in f
    traceback.print_stack(file=sys.stdout)
```

类似于 `format_exception()` 准备 `traceback`，`format_stack()` 以同样的方式准备栈轨迹。

```
import traceback
import sys
from pprint import pprint

from traceback_example import call_function
```

```
def f():
    return traceback.format_stack()
```

```
formatted_stack = call_function(f)
pprint(formatted_stack)
```

它返回一个串列表，每个串构成输出的一行。

```
$ python traceback_format_stack.py
```

```
[' File "traceback_format_stack.py", line 19, in <module>\n    form
atted_stack = call_function(f)\n',
  ' File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/tr
aceback_example.py", line 22, in call_function\n    return call_func
tion(f, recursion_level-1)\n',
  ' File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/tr
aceback_example.py", line 22, in call_function\n    return call_func
tion(f, recursion_level-1)\n',
  ' File "/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/traceback/tr
aceback_example.py", line 24, in call_function\n    return f()\n',
  ' File "traceback_format_stack.py", line 17, in f\n    return trac
eback.format_stack()\n']
```

extract_stack() 函数的工作类似于 extract_tb()。

```
import traceback
import sys
import os
from traceback_example import call_function
```

```
def f():
    return traceback.extract_stack()
```

```
stack = call_function(f)
for filename, linenum, funcname, source in stack:
    print '%-26s:%s "%s" in %s()' % \
        (os.path.basename(filename), linenum, source, funcname)
```

它还接受参数（这里没有显示），可以从栈中另一个位置开始，或者可以限制遍历的深度。

```
$ python traceback_extract_stack.py
```

```
traceback_extract_stack.py:19 "stack = call_function(f)" in <module>
()
traceback_example.py      :22 "return call_function(f, recursion_lev
el-1)" in call_function()
traceback_example.py      :22 "return call_function(f, recursion_lev
el-1)" in call_function()
traceback_example.py      :24 "return f()" in call_function()
traceback_extract_stack.py:17 "return traceback.extract_stack()" in
```

`f()`

参见:

`traceback` (<http://docs.python.org/lib/module-traceback.html>) 这个模块的标准库文档。

`sys` (17.2 节) `sys` 模块包括一些保存当前异常的单例对象。

`inspect` (18.4 节) `inspect` 模块包括另外一些函数来探查栈中的帧。

`cgitb` (16.5 节) 这是另一个用于美观地格式化 `traceback` 的模块。

16.5 cgitb——详细的 traceback 报告

作用: `cgitb` 能提供比 `traceback` 更为详细的 `traceback` 信息。

Python 版本: 2.2 及以后版本

`cgitb` 是标准库中一个很有价值的调试工具。它原先设计用来显示 Web 应用中的错误和调试信息。后来得到更新, 又包含了纯文本输出, 不过遗憾的是, 更新后并没有相应地改名。这个名字容易引起歧义, 以至于这个模块没有得到应有的关注, 原本它应该更为常用。

16.5.1 标准 traceback 转储

Python 的标准异常处理行为是向标准错误输出流打印一个 `traceback`, 并提供直至错误位置的调用栈。这个基本输出包含的信息通常足以了解异常的原因并做出修正。

```
def func2(a, divisor):
    return a / divisor
```

```
def func1(a, b):
    c = b - 5
    return func2(a, c)
```

```
func1(1, 5)
```

这个示例程序在 `func2()` 中有一个小错误。

```
$ python cgitb_basic_traceback.py
```

```
Traceback (most recent call last):
  File "cgitb_basic_traceback.py", line 17, in <module>
    func1(1, 5)
  File "cgitb_basic_traceback.py", line 15, in func1
    return func2(a, c)
  File "cgitb_basic_traceback.py", line 11, in func2
    return a / divisor
ZeroDivisionError: integer division or modulo by zero
```

16.5.2 启用详细 traceback

尽管基本 `traceback` 包括了足够的信息来发现错误, 不过启用 `cgitb` 将给出更多详细信息。

cgitb 将 `sys.excepthook` 替换为另一个函数，它能提供更丰富的 traceback。

```
import cgitb
cgitb.enable(format='text')
```

这个例子的错误报告比原先要丰富得多。会列出栈的每一帧，并提供以下信息：

- 源文件的完整路径，而不只是基名
- 栈中各个函数的参数值
- 错误路径中当前行周围的几行源代码上下文
- 导致错误的表达式中的变量值

由于能够访问错误栈中涉及的变量，这有助于找到栈中更高位置出现的一个逻辑错误，而不是生成实际异常的那一行代码。

```
$ python cgitb_local_vars.py
```

```
<type 'exceptions.ZeroDivisionError'>
Python 2.7: /Users/dhellmann/.virtualenvs/pymotw/bin/python
Sat Dec 4 12:59:15 2010
```

A problem occurred in a Python script. Here is the sequence of function calls leading up to the error, in the order they occurred.

```
/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_local_var
s.py in <module>()
    16 def func1(a, b):
    17     c = b - 5
    18     return func2(a, c)
    19
    20 func1(1, 5)
func1 = <function func1>

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_local_var
s.py in func1(a=1, b=5)
    16 def func1(a, b):
    17     c = b - 5
    18     return func2(a, c)
    19
    20 func1(1, 5)
global func2 = <function func2>
a = 1
c = 0
/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_local_var
s.py in func2(a=1, divisor=0)
    12
    13 def func2(a, divisor):
    14     return a / divisor
    15
```



```

16 def func1(a, b):
a = 1
divisor = 0
<type 'exceptions.ZeroDivisionError': integer division or modulo by
zero
  __class__ = <type 'exceptions.ZeroDivisionError'>
  __dict__ = {}
  __doc__ = 'Second argument to a division or modulo operation was
zero.'
...method references removed...
  args = ('integer division or modulo by zero',)
  message = 'integer division or modulo by zero'

```

The above is a description of an error in a Python program. Here is the original traceback:

```

Traceback (most recent call last):
  File "cgitb_local_vars.py", line 20, in <module>
    func1(1, 5)
  File "cgitb_local_vars.py", line 18, in func1
    return func2(a, c)
  File "cgitb_local_vars.py", line 14, in func2
    return a / divisor
ZeroDivisionError: integer division or modulo by zero

```

对于这个存在 `ZeroDivisionError` 异常的代码，显然问题是因为 `func1()` 中 `c` 值的计算而引入的，而不是因为在 `func2()` 中使用了这个值。

输出的最后还包括异常对象的完整细节（除了 `message` 以外可能还有其他属性，对调试会很有用），以及 `traceback` 转储的原始形式。

16.5.3 traceback 中的局部变量

`cgitb` 中的代码会检查栈帧中所使用的导致错误的变量，这些代码足够聪明，还可以计算对象属性并显示。

```

import cgitb
cgitb.enable(format='text', context=12)

class BrokenClass(object):
    """This class has an error.
    """

    def __init__(self, a, b):
        """Be careful passing arguments in here.
        """
        self.a = a
        self.b = b

```

```

        self.c = self.a * self.b
        # Really
        # long
        # comment
        # goes
        # here.
        self.d = self.a / self.b
    return

```

```
o = BrokenClass(1, 0)
```

如果一个函数或方法包括大量内联注释、空白符或其他代码，使它篇幅很长，倘若只有默认的 5 行上下文可能无法提供足够的指示。如果将函数体从显示的代码窗口取出，则没有足够的上下文来了解出现错误的位置。可以对 `cgibt` 使用一个更大的上下文值来解决这个问题。向 `enable()` 传入一个整数作为上下文 (`context`) 参数，控制为 `traceback` 的各行显示多少代码。

这个输出显示容易出错的代码与 `self.a` 和 `self.b` 有关。

```
$ python cgitb_with_classes.py | grep -v method
```

```

<type 'exceptions.ZeroDivisionError'>
Python 2.7: /Users/dhellmann/.virtualenvs/pymotw/bin/python
Sat Dec 4 12:59:16 2010

```

A problem occurred in a Python script. Here is the sequence of function calls leading up to the error, in the order they occurred.

```

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_with_classes.py in <module>()
20         self.a = a
21         self.b = b
22         self.c = self.a * self.b
23         # Really
24         # long
25         # comment
26         # goes
27         # here.
28         self.d = self.a / self.b
29         return
30
31 o = BrokenClass(1, 0)
o undefined
BrokenClass = <class '__main__.BrokenClass'>

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_with_classes.py in __init__(self=<__main__.BrokenClass object>, a=1, b=0)
20         self.a = a
21         self.b = b

```

```

22         self.c = self.a * self.b
23         # Really
24         # long
25         # comment
26         # goes
27         # here.
28         self.d = self.a / self.b
29         return
30
31 o = BrokenClass(1, 0)
self = <__main__.BrokenClass object>
self.d undefined
self.a = 1
self.b = 0
<type 'exceptions.ZeroDivisionError': integer division or modulo by
zero
    __class__ = <type 'exceptions.ZeroDivisionError'>
    __dict__ = {}
    __doc__ = 'Second argument to a division or modulo operation was
zero.'
...method references removed...
    args = ('integer division or modulo by zero',)
    message = 'integer division or modulo by zero'
The above is a description of an error in a Python program. Here is
the original traceback:

Traceback (most recent call last):
  File "cgitb_with_classes.py", line 31, in <module>
    o = BrokenClass(1, 0)
  File "cgitb_with_classes.py", line 28, in __init__
    self.d = self.a / self.b
ZeroDivisionError: integer division or modulo by zero

```

16.5.4 异常属性

除了每个栈帧的局部变量，cgitb 还会显示异常对象的所有属性。定制异常类型的额外属性会作为错误报告的一部分打印。

```

import cgitb
cgitb.enable(format='text')

class MyException(Exception):
    """Add extra properties to a special exception
    """

    def __init__(self, message, bad_value):
        self.bad_value = bad_value
        Exception.__init__(self, message)

```

```
return
```

```
raise MyException('Normal message', bad_value=99)
```

在这个例子中，除了标准的 message 和 args 值外，还包含 bad_value 属性。

```
$ python cgitb_exception_properties.py
```

```
<class '__main__.MyException'>
```

```
Python 2.7: /Users/dhellmann/.virtualenvs/pymotw/bin/python
```

```
Sat Dec 4 12:59:16 2010
```

A problem occurred in a Python script. Here is the sequence of function calls leading up to the error, in the order they occurred.

```
/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_exception_properties.py in <module>()
```

```
18         self.bad_value = bad_value
```

```
19         Exception.__init__(self, message)
```

```
20         return
```

```
21
```

```
22 raise MyException('Normal message', bad_value=99)
```

```
MyException = <class '__main__.MyException'>
```

```
bad_value undefined
```

```
<class '__main__.MyException'>: Normal message
```

```
__class__ = <class '__main__.MyException'>
```

```
__dict__ = {'bad_value': 99}
```

```
__doc__ = 'Add extra properties to a special exception\n
```

```
__module__ = '__main__'
```

```
...method references removed...
```

```
args = ('Normal message',)
```

```
bad_value = 99
```

```
message = 'Normal message'
```

The above is a description of an error in a Python program. Here is the original traceback:

```
Traceback (most recent call last):
```

```
File "cgitb_exception_properties.py", line 22, in <module>
```

```
    raise MyException('Normal message', bad_value=99)
```

```
MyException: Normal message
```

16.5.5 HTML 输出

由于 cgitb 原来是为了处理 Web 应用中用的异常开发的，如果不提到原来的 HTML 输出格式，就不能算是个完整的讨论。前面的例子都显示了纯文本输出。为了生成 HTML 输出，要省略格式 (format) 参数 (或者指定 “html”)。大多数现代 Web 应用都使用一个包含错误报告功能的框架来构造，所以很大程度上讲，HTML 格式已经过时。

16.5.6 记录 traceback

很多情况下，将 traceback 细节打印到标准错误输出是最佳的解决方案。不过，在生产系统中，更好的做法是记录错误日志。enable() 函数包括一个可选参数 logdir，用来启用错误日志。提供一个目录名时，每个异常会记入指定目录该异常自己的文件中。

```
import cgitb
import os

cgitb.enable(logdir=os.path.join(os.path.dirname(__file__), 'LOGS'),
             display=False,
             format='text',
             )

def func(a, divisor):
    return a / divisor

func(1, 0)
```

尽管“抑制”了错误显示，还是会打印一个消息来指出要到哪里查找错误日志。

```
$ python cgitb_log_exception.py
```

```
<p>A problem occurred in a Python script.
<p> /Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/LOGS/tmpy2v8
NM.txt contains the description of this error.
```

```
$ ls LOGS
```

```
tmpy2v8NM.txt
```

```
$ cat LOGS/*.txt
```

```
<type 'exceptions.ZeroDivisionError'>
Python 2.7: /Users/dhellmann/.virtualenvs/pymotw/bin/python
Sat Dec 4 12:59:15 2010
```

```
A problem occurred in a Python script. Here is the sequence of
function calls leading up to the error, in the order they occurred.
```

```
/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_log_excep
tion.py in <module>()
 17
 18 def func(a, divisor):
 19     return a / divisor
 20
 21 func(1, 0)
func = <function func>
```

```

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/cgitb/cgitb_log_excep
tion.py in func(a=1, divisor=0)
    17
    18 def func(a, divisor):
    19     return a / divisor
    20
    21 func(1, 0)
a = 1
divisor = 0
<type 'exceptions.ZeroDivisionError': integer division or modulo by
zero
    __class__ = <type 'exceptions.ZeroDivisionError'>
    __delattr__ = <method-wrapper '__delattr__' of
exceptions.ZeroDivisionError object>
    __dict__ = {}
    __doc__ = 'Second argument to a division or modulo operation was
zero.'
    __format__ = <built-in method __format__ of
exceptions.ZeroDivisionError object>
    __getattr__ = <method-wrapper '__getattr__' of
exceptions.ZeroDivisionError object>
    __getitem__ = <method-wrapper '__getitem__' of
exceptions.ZeroDivisionError object>
    __getslice__ = <method-wrapper '__getslice__' of
exceptions.ZeroDivisionError object>
    __hash__ = <method-wrapper '__hash__' of
exceptions.ZeroDivisionError object>
    __init__ = <method-wrapper '__init__' of
exceptions.ZeroDivisionError object>
    __new__ = <built-in method __new__ of type object>
    __reduce__ = <built-in method __reduce__ of
exceptions.ZeroDivisionError object>
    __reduce_ex__ = <built-in method __reduce_ex__ of
exceptions.ZeroDivisionError object>
    __repr__ = <method-wrapper '__repr__' of
exceptions.ZeroDivisionError object>
    __setattr__ = <method-wrapper '__setattr__' of
exceptions.ZeroDivisionError object>
    __setstate__ = <built-in method __setstate__ of
exceptions.ZeroDivisionError object>
    __sizeof__ = <built-in method __sizeof__ of
exceptions.ZeroDivisionError object>
    __str__ = <method-wrapper '__str__' of
exceptions.ZeroDivisionError object>
    __subclasshook__ = <built-in method __subclasshook__ of type
object>
    __unicode__ = <built-in method __unicode__ of

```

```
exceptions.ZeroDivisionError object>
args = ('integer division or modulo by zero',)
message = 'integer division or modulo by zero'
```

The above is a description of an error in a Python program. Here is the original traceback:

```
Traceback (most recent call last):
  File "cgitb_log_exception.py", line 21, in <module>
    func(1, 0)
  File "cgitb_log_exception.py", line 19, in func
    return a / divisor
ZeroDivisionError: integer division or modulo by zero
```

参见:

cgitb (<http://docs.python.org/library/cgitb.html>) 这个模块的标准库文档。

traceback (16.4 节) 处理 traceback 的标准库模块。

inspect (18.4 节) inspect 模块包含更多函数来检查栈。

sys (17.2 节) sys 模块允许访问当前异常值和出现异常时所调用的 excepthook 处理程序。

Improved Traceback Module(<http://thread.gmane.org/gmane.comp.python.devel/110326>) 讨论了 Python 开发邮件列表中关于 traceback 模块的改进以及其他开发人员局部使用的相关改进。

16.6 pdb——交互式调试工具

作用: Python 的交互式调试工具。

Python 版本: 1.4 及以后版本

pdb 为 Python 程序实现了一个交互式调试环境。它包括一些特性, 可以暂停程序, 查看变量值, 逐步监视程序执行, 使你能了解程序具体做了什么, 并查找逻辑中存在的 bug。

16.6.1 启动调试工具

使用 pdb 的第一步是让解释器在适当时进入调试工具。为此有很多不同的方法, 取决于起始条件和所要调试的内容。

从命令行运行

使用调试工具的最直接方法是从命令行运行调试工具, 提供程序作为输入, 使它知道要运行什么。

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
```

```

7  class MyObj(object):
8
9      def __init__(self, num_loops):
10         self.count = num_loops
11
12     def go(self):
13         for i in range(self.count):
14             print i
15         return
16
17 if __name__ == '__main__':
18     MyObj(5).go()

```

从命令行运行调试工具时，它会加载源文件，并在找到的第一条语句处停止执行。在这里，它会在第 7 行类 `MyObj` 的定义之前停止。

```

$ python -m pdb pdb_script.py

> .../pdb_script.py(7)<module>()
-> class MyObj(object):
(Pdb)

```

注意：通常，pdb 打印一个文件名时会在输出中包含各模块的完整路径。为了保证例子的简洁性，这一节示例输出中的路径被替换为一个省略号 (...).

在解释器中运行

很多 Python 开发人员开发模块的较早版本时会使用交互式解释器，因为这样他们能反复实验，而不用像创建独立脚本时那样，需要完整的保存 / 运行 / 重复周期。要在一个交互式解释器中运行调试工具，可以使用 `run()` 或 `runeval()`。

```

$ python

Python 2.7 (r27:82508, Jul 3 2010, 21:12:11)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb_script
>>> import pdb
>>> pdb.run('pdb_script.MyObj(5).go()')
> <string>(1)<module>()
(Pdb)

```

`run()` 的参数是一个串表达式，可以由 Python 解释器计算。调试工具会进行解析，然后在计算第一个表达式之前暂停执行。这里介绍的调试工具命令可以用来导航和控制执行。

从程序中运行

前面的两个例子都是从程序一开始就启动调试工具。对于一个长时间运行的进程，问题可能出现在程序执行较后的时刻，更方便的做法是在程序中使用 `set_trace()` 启用调试工具。


```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  import pdb
8
9  class MyObj(object):
10
11     def __init__(self, num_loops):
12         self.count = num_loops
13
14     def go(self):
15         for i in range(self.count):
16             pdb.set_trace()
17             print i
18         return
19
20 if __name__ == '__main__':
21     MyObj(5).go()

```

示例脚本的第 16 行在执行到该点时触发调试工具。

```
$ python ./pdb_set_trace.py
```

```

> .../pdb_set_trace.py(17)go()
-> print i
(Pdb)

```

`set_trace()` 只是一个 Python 函数，所以可以在程序中任意位置调用。这样就可以根据程序中的条件进入调试工具，包括从一个异常处理程序进入，或者通过一个控制语句的特定分支进入。

失败后运行

在程序终止后调试失败称为事后剖析调试 (post-mortem debugging)。pdb 通过 `pm()` 和 `post_mortem()` 函数支持事后剖析调试。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  class MyObj(object):
8
9     def __init__(self, num_loops):
10         self.count = num_loops
11
12     def go(self):

```

```

13         for i in range(self.num_loops):
14             print i
15         return

```

在这里第 13 行上不正确的属性名触发了一个 `AttributeError` 异常，导致执行停止，`pm()` 查找活动 `traceback`，在调用栈中出现异常的位置启动调试工具。

```

$ python

Python 2.7 (r27:82508, Jul  3 2010, 21:12:11)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pdb_post_mortem import MyObj
>>> MyObj(5).go()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pdb_post_mortem.py", line 13, in go
    for i in range(self.num_loops):
AttributeError: 'MyObj' object has no attribute 'num_loops'
>>> import pdb
>>> pdb.pm()
> .../pdb_post_mortem.py(13)go()
-> for i in range(self.num_loops):
(Pdb)

```

16.6.2 控制调试工具

调试工具的接口是一个很小的命令语言，允许在调用栈中移动、检查和修改变量的值，以及控制调试工具如何执行程序。这个交互式调试工具使用 `readline` 来接受命令。输入一个空行会再次重新运行之前的命令，除非是一个 `list` 操作。

导航执行栈

调试工具运行的任何时刻，使用 `where`（缩写为 `w`）可以得出正在执行哪一行，以及程序在调用栈的哪个位置。在这里，正在执行模块 `pdb_set_trace.py` 的 `go()` 方法中的第 17 行。

```

$ python pdb_set_trace.py

> .../pdb_set_trace.py(17)go()
-> print i
(Pdb) where
.../pdb_set_trace.py(21)<module>()
-> MyObj(5).go()
> .../pdb_set_trace.py(17)go()
-> print i

```

要围绕当前位置添加更多上下文，可以使用 `list` (`l`)。

```

(Pdb) list
12             self.count = num_loops

```

```

13
14     def go(self):
15         for i in range(self.count):
16             pdb.set_trace()
17 ->         print i
18         return
19
20 if __name__ == '__main__':
21     MyObj(5).go()
[EOF]
(Pdb)

```

默认会列出当前行周围共 11 行（前面 5 行，后面 5 行）。如果使用 `list` 并提供一个数值参数，则会列出指定行（而不是当前行）周围的 11 行。

```

(Pdb) list 14
9     class MyObj(object):
10
11         def __init__(self, num_loops):
12             self.count = num_loops
13
14         def go(self):
15             for i in range(self.count):
16                 pdb.set_trace()
17 ->                 print i
18             return
19

```

如果 `list` 接受两个参数，会把它们解释为输出中包含的第一行和最后一行。

```

(Pdb) list 5, 19
5     #
6
7     import pdb
8
9     class MyObj(object):
10
11         def __init__(self, num_loops):
12             self.count = num_loops
13
14         def go(self):
15             for i in range(self.count):
16                 pdb.set_trace()
17 ->                 print i
18             return
19

```

可以使用 `up` 和 `down` 在当前调用栈的帧之间移动。`up`（缩写为 `u`）向栈中较早的帧移动。`down`（缩写为 `d`）则移向较新的帧。

```
(Pdb) up
> .../pdb_set_trace.py(21)<module>()
-> MyObj(5).go()
```

```
(Pdb) down
> .../pdb_set_trace.py(17)go()
-> print i
```

每次在栈中上移或下移时，调试工具会打印当前位置，格式与 `where` 生成的格式相同。

检查栈中的变量

栈中的各帧维护一组变量，包括所执行函数的局部值和全局状态信息。pdb 提供了多种方法来检查这些变量的内容。

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  import pdb
8
9  def recursive_function(n=5, output='to be printed'):
10     if n > 0:
11         recursive_function(n-1)
12     else:
13         pdb.set_trace()
14         print output
15     return
16
17 if __name__ == '__main__':
18     recursive_function()
```

`args` 命令（缩写为 `a`）会打印当前帧中活动函数的所有参数。这个例子还使用了一个递归函数，以显示由 `where` 打印一个更深的栈时会得到什么结果。

```
$ python pdb_function_arguments.py

> .../pdb_function_arguments.py(14)recursive_function()
-> return
(Pdb) where
.../pdb_function_arguments.py(17)<module>()
-> recursive_function()
.../pdb_function_arguments.py(11)recursive_function()
-> recursive_function(n-1)
.../pdb_function_arguments.py(11)recursive_function()
-> recursive_function(n-1)
.../pdb_function_arguments.py(11)recursive_function()
-> recursive_function(n-1)
```

```

.../pdb_function_arguments.py(11)recursive_function()
-> recursive_function(n-1)
.../pdb_function_arguments.py(11)recursive_function()
-> recursive_function(n-1)
> .../pdb_function_arguments.py(14)recursive_function()
-> return

```

```

(Pdb) args
n = 0
output = to be printed

```

```

(Pdb) up
> .../pdb_function_arguments.py(11)recursive_function()
-> recursive_function(n-1)

```

```

(Pdb) args
n = 1
output = to be printed

```

```

(Pdb)

```

p 命令会计算作为参数给定的一个表达式，并打印其结果。也可以使用 Python 的 print 语句，不过要把它传至解释器来执行，而不是在调试工具中作为一个命令运行。

```

(Pdb) p n
1

```

```

(Pdb) print n
1

```

类似地，在一个表达式前面加上前缀 ! 就会把它传递到 Python 解释器进行计算。这个特性可以用来执行任意的 Python 语句，包括修改变量。下面这个例子在允许调试工具继续运行程序之前修改了 output 的值。set_trace() 调用后的下一条语句打印出 output 的值，会显示修改后的值。

```

$ python pdb_function_arguments.py

> .../pdb_function_arguments.py(14)recursive_function()
-> print output

(Pdb) !output
'to be printed'

(Pdb) !output='changed value'

(Pdb) continue
changed value

```

对于更复杂的值，如嵌套数据结构或大型数据结构，要使用 pp 以“完美打印”格式进行打印。下面这个程序从一个文件读取多个文本行。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  import pdb
8
9  with open('lorem.txt', 'rt') as f:
10     lines = f.readlines()
11
12  pdb.set_trace()

```

用 `p` 打印变量 `lines` 时，得到的输出很难读，因为它的换行很拙劣。pp 使用 `pprint` 格式化值以便美观地打印。

```
$ python pdb_pp.py
```

```
--Return--
```

```
> .../pdb_pp.py(12)<module>()->None
```

```
-> pdb.set_trace()
```

```
(Pdb) p lines
```

```
['Lorem ipsum dolor sit amet, consectetur adipiscing elit. \n',
 'Donec egestas, enim et consete
tuer ullamcorper, lectus \n', 'ligula rutrum leo, a elementum el
it tortor eu quam.\n']
```

```
(Pdb) pp lines
```

```
['Lorem ipsum dolor sit amet, consectetur adipiscing elit. \n',
 'Donec egestas, enim et consectetur ullamcorper, lectus \n',
 'ligula rutrum leo, a elementum elit tortor eu quam.\n']
```

```
(Pdb)
```

单步执行程序

除了程序暂停时在调用栈中上下导航外，还可以在进入调试工具那一点之后单步执行程序。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  import pdb
8
9  def f(n):
10     for i in range(n):
11         j = i * n

```

```

12         print i, j
13     return
14
15 if __name__ == '__main__':
16     pdb.set_trace()
17     f(5)

```

使用 `step` 执行当前行，然后在下一个执行点停止——这可能是所调用的函数中的第一条语句，也可能是当前函数的下一行语句。

```
$ python pdb_step.py
```

```
> .../pdb_step.py(17)<module>()
-> f(5)
```

解释器会在 `set_trace()` 调用处暂停，将控制交给调试工具。第一步会导致执行进入 `f()`。

```
(Pdb) step
--Call--
> .../pdb_step.py(9)f()
-> def f(n):
```

再执行一步会执行到 `f()` 的第一行，并开始循环。

```
(Pdb) step
> .../pdb_step.py(10)f()
-> for i in range(n):
```

执行下一步会移动到循环中的第一行，即定义 `j` 的代码。

```
(Pdb) step
> .../pdb_step.py(11)f()
-> j = i * n
(Pdb) p i
0
```

`i` 的值为 0，所以再执行一步后，`j` 的值应当也是 0。

```
(Pdb) step
> .../pdb_step.py(12)f()
-> print i, j
```

```
(Pdb) p j
0
```

```
(Pdb)
```

像这样一次执行一步，如果在出现错误那一点之前需要执行很多代码，或者如果需要反复调用相同的函数，就会变得很麻烦。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #

```

```
4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5 #
6
7 import pdb
8
9 def calc(i, n):
10     j = i * n
11     return j
12
13 def f(n):
14     for i in range(n):
15         j = calc(i, n)
16         print i, j
17     return
18
19 if __name__ == '__main__':
20     pdb.set_trace()
21     f(5)
```

在这个例子中，`calc()` 没有错误，所以在 `f()` 的循环中每次调用它时如果都单步跟踪，执行时会显示 `calc()` 的所有代码行，这对有用的输出会造成干扰。

```
$ python pdb_next.py
```

```
> .../pdb_next.py(21)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(13)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(14)f()
-> for i in range(n):

(Pdb) step
> .../pdb_next.py(15)f()
-> j = calc(i, n)

(Pdb) step
--Call--
> .../pdb_next.py(9)calc()
-> def calc(i, n):

(Pdb) step
> .../pdb_next.py(10)calc()
-> j = i * n
```




```
(Pdb) step
> .../pdb_next.py(11)calc()
-> return j

(Pdb) step
--Return--
> .../pdb_next.py(11)calc()->0
-> return j

(Pdb) step
> .../pdb_next.py(16)f()
-> print i, j

(Pdb) step
0 0
```

`next` 命令有些类似 `step`，不过不会从正在执行的语句进入所调用的函数。实际上，它会用一个操作完成整个函数调用，而直接进入当前函数的下一条语句。

```
> .../pdb_next.py(14)f()
-> for i in range(n):
(Pdb) step
> .../pdb_next.py(15)f()
-> j = calc(i, n)

(Pdb) next
> .../pdb_next.py(16)f()
-> print i, j

(Pdb)
```

`until` 命令类似于 `next`，只不过它会继续执行，直至执行到同一个函数中行号大于当前值的一行。例如，这说明，`until` 可以用于跳过循环末尾。

```
$ python pdb_next.py

> .../pdb_next.py(21)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(13)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(14)f()
-> for i in range(n):

(Pdb) step
```



```
> .../pdb_next.py(15)f()
-> j = calc(i, n)

(Pdb) next
> .../pdb_next.py(16)f()
-> print i, j
(Pdb) until
0 0
1 5
2 10
3 15
4 20
> .../pdb_next.py(17)f()
-> return
```

(Pdb)

运行 until 命令之前, 当前行为 16, 即循环的最后一行。运行 until 之后, 执行位于第 17 行, 循环已经结束。

return 命令也是绕开函数部分的一个捷径。它会继续执行, 直至函数将要执行一个 return 语句, 然后它会暂停, 从而在函数返回之前有时间查看返回值。

```
$ python pdb_next.py

> .../pdb_next.py(21)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(13)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(14)f()
-> for i in range(n):

(Pdb) return
0 0
1 5
2 10
3 15
4 20
--Return--
> .../pdb_next.py(17)f()->None
-> return

(Pdb)
```



16.6.3 断点

随着程序越来越长，即使使用 `next` 和 `until` 也会变得很慢，很繁琐。不用手动地单步跟踪程序，一种更好的解决方案是让它正常运行，直至达到某一点，调试工具要在这一点中断执行。`set_trace()` 可以启动调试工具，不过程序中只有一个要暂停的点时这才适用。更方便的做法是通过调试工具运行程序，但是使用断点 (breakpoint) 提前告诉调试工具在哪里停止。调试工具会监视程序，到达断点描述的位置时，程序会在执行那一行之前暂停。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  def calc(i, n):
8      j = i * n
9      print 'j =', j
10     if j > 0:
11         print 'Positive!'
12     return j
13
14 def f(n):
15     for i in range(n):
16         print 'i =', i
17         j = calc(i, n)
18     return
19
20 if __name__ == '__main__':
21     f(5)

```

`break` 命令有很多选项用来设置断点，包括要暂停处理的行号、文件和函数。要在当前文件的一个特定行设置断点，可以使用 `break lineno`。

```

$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break 11
Breakpoint 1 at .../pdb_break.py:11

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(11)calc()
-> print 'Positive!'

```



```
(Pdb)
```

命令 `continue` 告诉调试工具继续运行程序，直到到达下一个断点。在这里，它会运行完 `f()` 中 `for` 循环的第一次迭代，第二次迭代期间在 `calc()` 中停止。

可以指定函数名而不是一个行号，把断点设置到一个函数的第一行。下面这个例子显示了为 `calc()` 函数增加一个断点会发生什么。

```
$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:7

(Pdb) continue
i = 0
> .../pdb_break.py(8)calc()
-> j = i * n

(Pdb) where
.../pdb_break.py(21)<module>()
-> f(5)
.../pdb_break.py(17)f()
-> j = calc(i, n)
> .../pdb_break.py(8)calc()
-> j = i * n

(Pdb)
```

要在另一个文件中指定一个断点，可以在行或函数参数前加一个文件名前缀。

```
1 #!/usr/bin/env python
2 # encoding: utf-8
3
4 from pdb_break import f
5
6 f(5)
```

在这里，开始主程序 `pdb_break_remote.py` 之后，为 `pdb_break.py` 的第 11 行设置了一个断点。

```
$ python -m pdb pdb_break_remote.py

> .../pdb_break_remote.py(4)<module>()
-> from pdb_break import f
(Pdb) break pdb_break.py:11
Breakpoint 1 at .../pdb_break.py:11

(Pdb) continue
i = 0
```

```
j = 0
i = 1
j = 5
> ../pdb_break.py(11)calc()
-> print 'Positive!'
```

(Pdb)

文件名可以是源文件的完整路径，也可以是相对于 sys.path 上某个文件的相对路径。

要列出当前设置的断点，可以使用 **break** 而不带任何参数。输出包括文件和各个断点的行号，以及多少次遇到这个断点的有关信息。

```
$ python -m pdb pdb_break.py
```

```
> ../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break 11
Breakpoint 1 at ../pdb_break.py:11
```

```
(Pdb) break
Num Type      Disp Enb  Where
1  breakpoint keep yes   at ../pdb_break.py:11
```

```
(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> ../pdb/pdb_break.py(11)calc()
-> print 'Positive!'
```

```
(Pdb) continue
Positive!
i = 2
j = 10
> ../pdb_break.py(11)calc()
-> print 'Positive!'
```

```
(Pdb) break
Num Type      Disp Enb  Where
1  breakpoint keep yes   at ../pdb_break.py:11
breakpoint already hit 2 times
```

(Pdb)

管理断点

增加各个新断点时，会为它指定一个数值标识符。这些 id 号用于交互式地启用、禁用和删

除断点。用 `disable` 关闭一个断点时，会告诉调试工具到达该行时不要停止。这个断点仍会记住，但是会将其忽略。

```
$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:7

(Pdb) break 11
Breakpoint 2 at .../pdb_break.py:11

(Pdb) break
Num Type          Disp Enb  Where
1  breakpoint      keep yes  at .../pdb_break.py:7
2  breakpoint      keep yes  at .../pdb_break.py:11

(Pdb) disable 1

(Pdb) break
Num Type          Disp Enb  Where
1  breakpoint      keep no   at .../pdb_break.py:7
2  breakpoint      keep yes  at .../pdb_break.py:11

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(11)calc()
-> print 'Positive!'

(Pdb)
```

下一个调试会话在程序中设置两个断点，然后禁用其中一个。程序会一直运行，直至遇到留下的那个断点，然后在执行继续之前用 `enable` 把另一个断点打开。

```
$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:7

(Pdb) break 16
Breakpoint 2 at .../pdb_break.py:16
```

```
(Pdb) disable 1

(Pdb) continue
> .../pdb_break.py(16)f()
-> print 'i =', i

(Pdb) list
11             print 'Positive!'
12             return j
13
14     def f(n):
15         for i in range(n):
16 B->             print 'i =', i
17                 j = calc(i, n)
18             return
19
20     if __name__ == '__main__':
21         f(5)

(Pdb) continue
i = 0
j = 0
> .../pdb_break.py(16)f()
-> print 'i =', i

(Pdb) list
11             print 'Positive!'
12             return j
13
14     def f(n):
15         for i in range(n):
16 B->             print 'i =', i
17                 j = calc(i, n)
18             return
19
20     if __name__ == '__main__':
21         f(5)

(Pdb) p i
1

(Pdb) enable 1

(Pdb) continue
i = 1
> .../pdb_break.py(8)calc()
-> j = i * n
```



```
(Pdb) list
3      #
4      # Copyright (c) 2010 Doug Hellmann.  All rights reserved.
5      #
6
7 B    def calc(i, n):
8 ->        j = i * n
9          print 'j =', j
10         if j > 0:
11             print 'Positive!'
12         return j
13
```

```
(Pdb)
```

list 的输出中带 B 前缀的行显示了程序中哪里设置了断点（第 7 行和第 16 行）。

可以使用 clear 完全删除一个断点。

```
$ python -m pdb pdb_break.py
```

```
> .../pdb_break.py(7)<module>()
```

```
-> def calc(i, n):
```

```
(Pdb) break calc
```

```
Breakpoint 1 at .../pdb_break.py:7
```

```
(Pdb) break 11
```

```
Breakpoint 2 at .../pdb_break.py:11
```

```
(Pdb) break 16
```

```
Breakpoint 3 at .../pdb_break.py:16
```

```
(Pdb) break
```

Num	Type	Disp	Enb	Where
1	breakpoint	keep	yes	at .../pdb_break.py:7
2	breakpoint	keep	yes	at .../pdb_break.py:11
3	breakpoint	keep	yes	at .../pdb_break.py:16

```
(Pdb) clear 2
```

```
Deleted breakpoint 2
```

```
(Pdb) break
```

Num	Type	Disp	Enb	Where
1	breakpoint	keep	yes	at .../pdb_break.py:7
3	breakpoint	keep	yes	at .../pdb_break.py:16

```
(Pdb)
```

其他断点仍保留原来的标识符，不会重新编号。

临时断点

程序第一次执行到临时断点时会将它自动清除。通过使用临时断点，可以很快到达程序流中的特定位置，这与常规断点一样，只是它会立即清除。不过，如果这部分程序反复运行，临时断点不会干扰后续执行。

```
$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) tbreak 11
Breakpoint 1 at .../pdb_break.py:11

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
Deleted breakpoint 1
> .../pdb_break.py(11)calc()
-> print 'Positive!'

(Pdb) break

(Pdb) continue
Positive!
i = 2
j = 10
Positive!
i = 3
j = 15
Positive!
i = 4
j = 20
Positive!
The program finished and will be restarted
> .../pdb_break.py(7)<module>()
-> def calc(i, n):

(Pdb)
```

程序第一次到达第 11 行时，会将断点删除，在程序完成之前不会再停止执行。

条件断点

可以对断点应用一些规则，仅当条件满足时执行才停止。相对于手动地启用和禁用断点，使用条件断点可以对调试工具如何暂停程序提供更精细的控制。可以用两种方式设置条件断点。第一种是使用 `break` 指定设置断点时的条件。

```
$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break 9, j>0
Breakpoint 1 at .../pdb_break.py:9

(Pdb) break
Num Type          Disp Enb   Where
1  breakpoint      keep yes    at .../pdb_break.py:9
    stop only if j>0

(Pdb) continue
i = 0
j = 0
i = 1
> .../pdb_break.py(9)calc()
-> print 'j =', j

(Pdb)
```

条件参数必须是一个表达式，要使用定义断点的栈帧中可见的值。如果表达式计算为 `true`，则在断点处停止执行。

还可以使用 `condition` 命令对一个现有的断点应用条件。参数是断点 `id` 和表达式。

```
$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break 9
Breakpoint 1 at .../pdb_break.py:9

(Pdb) break
Num Type          Disp Enb   Where
1  breakpoint      keep yes    at .../pdb_break.py:9

(Pdb) condition 1 j>0

(Pdb) break
Num Type          Disp Enb   Where
1  breakpoint      keep yes    at .../pdb_break.py:9
    stop only if j>0

(Pdb)
```

忽略断点

有些程序包含有循环，或者使用了大量相同函数的递归调用，通过在执行中“前跳”

(skipping ahead) 可以更容易地调试这些程序，而不是监视每一个调用或断点。ignore 命令告诉调试工具跳过一个断点而不停止。每次处理遇到这个断点时，它会将忽略计数器递减。当这个计数器为 0 时，会再次启用这个断点。

```
$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break 17
Breakpoint 1 at .../pdb_break.py:17

(Pdb) continue
i = 0
> .../pdb_break.py(17)f()
-> j = calc(i, n)

(Pdb) next
j = 0
> .../pdb_break.py(15)f()
-> for i in range(n):

(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.
(Pdb) break
Num Type          Disp Enb   Where
1  breakpoint    keep yes   at .../pdb_break.py:17
    ignore next 2 hits
    breakpoint already hit 1 time

(Pdb) continue
i = 1
j = 5
Positive!
i = 2
j = 10
Positive!
i = 3
> .../pdb_break.py(17)f()
-> j = calc(i, n)

(Pdb) break
Num Type          Disp Enb   Where
1  breakpoint    keep yes   at .../pdb_break.py:17
    breakpoint already hit 4 times
```

显式地重新设置忽略计数器为 0，可以立即再次启用这个断点。

```
$ python -m pdb pdb_break.py
```



```

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break 17
Breakpoint 1 at .../pdb_break.py:17

(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.

(Pdb) break
Num Type          Disp Enb   Where
1  breakpoint      keep yes   at .../pdb_break.py:17
    ignore next 2 hits

(Pdb) ignore 1 0
Will stop next time breakpoint 1 is reached.

(Pdb) break
Num Type          Disp Enb   Where
1  breakpoint      keep yes   at .../pdb_break.py:17

```

在断点触发动作

除了纯交互式模式，pdb 还支持基本脚本模式。通过使用 `commands`，可以在遇到一个特定断点时执行一系列解释器命令，也包括 Python 语句。运行 `commands` 并提供断点号作为参数，调试工具提示符会变为 `(com)`。一次输入一个命令，用 `end` 结束命令列表来保存脚本，然后返回到主调试工具提示符。

```

$ python -m pdb pdb_break.py

> .../pdb_break.py(7)<module>()
-> def calc(i, n):
(Pdb) break 9
Breakpoint 1 at .../pdb_break.py:9

(Pdb) commands 1
(com) print 'debug i =', i
(com) print 'debug j =', j
(com) print 'debug n =', n
(com) end

(Pdb) continue
i = 0
debug i = 0
debug j = 0
debug n = 5
> .../pdb_break.py(9)calc()
-> print 'j =', j

```



```

(Pdb) continue
j = 0
i = 1
debug i = 1
debug j = 5
debug n = 5
> ../pdb_break.py(9)calc()
-> print 'j =', j

```

```
(Pdb)
```

如果代码使用了大量数据结构或变量，这个特性对于调试这些代码尤其有用，因为可以让调试工具自动地打印出所有值，而不是每次遇到断点时手动地处理。

16.6.4 改变执行流

`jump` 命令可以在运行时改变程序流，而不修改代码。它可以向前跳不再运行某些代码，或者向后跳再次运行某些代码。下面的示例程序会生成一个数字列表。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  def f(n):
8      result = []
9      j = 0
10     for i in range(n):
11         j = i * n + j
12         j += n
13         result.append(j)
14     return result
15
16 if __name__ == '__main__':
17     print f(5)

```

不加干扰地运行时，输出是整除 5 的递增数字组成的一个序列。

```
$ python pdb_jump.py
```

```
[5, 15, 30, 50, 75]
```

前跳

前跳（Jump Ahead）会把执行点移至当前位置之后，而不再执行二者之间的任何语句。在下面这个例子中，由于跳过了第 13 行，`j` 的值并不递增，后面所有依赖于它的值都会稍小一点。

```
$ python -m pdb pdb_jump.py
```

```
> .../pdb_jump.py(7)<module>()
-> def f(n):
(Pdb) break 12
Breakpoint 1 at .../pdb_jump.py:12

(Pdb) continue
> .../pdb_jump.py(12)f()
-> j += n

(Pdb) p j
0

(Pdb) step
> .../pdb_jump.py(13)f()
-> result.append(j)

(Pdb) p j
5

(Pdb) continue
> .../pdb_jump.py(12)f()
-> j += n

(Pdb) jump 13
> .../pdb_jump.py(13)f()
-> result.append(j)

(Pdb) p j
10

(Pdb) disable 1

(Pdb) continue
[5, 10, 25, 45, 70]

The program finished and will be restarted
> .../pdb_jump.py(7)<module>()
-> def f(n):
(Pdb)
```

后跳

也可以后跳，让程序执行之前已经执行过的语句，使它再次运行。在这里，j 值多递增一次，得到的序列中，数都比原本的值大一些。

```
$ python -m pdb pdb_jump.py

> .../pdb_jump.py(7)<module>()
```

```

-> def f(n):
(Pdb) break 13
Breakpoint 1 at .../pdb_jump.py:13

(Pdb) continue
> .../pdb_jump.py(13)f()
-> result.append(j)

(Pdb) p j
5

(Pdb) jump 12
> .../pdb_jump.py(12)f()
-> j += n

(Pdb) continue
> .../pdb_jump.py(13)f()
-> result.append(j)

(Pdb) p j
10

(Pdb) disable 1

(Pdb) continue
[10, 20, 35, 55, 80]

The program finished and will be restarted
> .../pdb_jump.py(7)<module>()
-> def f(n):
(Pdb)

```

非法跳转

跳入和跳出某些流控制语句，这样很危险，而且不确定，因此调试工具不允许这么做。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  def f(n):
8      if n < 0:
9          raise ValueError('Invalid n: %s' % n)
10     result = []
11     j = 0
12     for i in range(n):

```

```
13         j = i * n + j
14         j += n
15         result.append(j)
16     return result
17
18
19 if __name__ == '__main__':
20     try:
21         print f(5)
22     finally:
23         print 'Always printed'
24
25     try:
26         print f(-5)
27     except:
28         print 'There was an error'
29     else:
30         print 'There was no error'
31
32     print 'Last statement'
```

可以使用 `jump` 进入一个函数，但是参数未定义，代码往往不能工作。

```
$ python -m pdb pdb_no_jump.py
```

```
> .../pdb_no_jump.py(7)<module>()
-> def f(n):
(Pdb) break 21
Breakpoint 1 at .../pdb_no_jump.py:21
```

```
(Pdb) jump 8
> .../pdb_no_jump.py(8)<module>()
-> if n < 0:

(Pdb) p n
*** NameError: NameError("name 'n' is not defined",)

(Pdb) args

(Pdb)
```

`jump` 不会进入类似 `for` 循环或 `try:except` 语句等代码块的中间。

```
$ python -m pdb pdb_no_jump.py
```

```
> .../pdb_no_jump.py(7)<module>()
-> def f(n):
(Pdb) break 21
Breakpoint 1 at .../pdb_no_jump.py:21
```




```
(Pdb) continue
> .../pdb_no_jump.py(21)<module>()
-> print f(5)

(Pdb) jump 26
*** Jump failed: can't jump into the middle of a block

(Pdb)
```

finally 块中的代码必须全部执行，所以 jump 不会离开 finally 块。

```
$ python -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(7)<module>()
-> def f(n):
(Pdb) break 23
Breakpoint 1 at .../pdb_no_jump.py:23

(Pdb) continue
[5, 15, 30, 50, 75]
> .../pdb_no_jump.py(23)<module>()
-> print 'Always printed'

(Pdb) jump 25
*** Jump failed: can't jump into or out of a 'finally' block

(Pdb)
```

最基本的限制是，跳转要受调用栈最底层帧的约束。在栈中上移检查变量之后，不能在这一点上改变执行流。

```
$ python -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(7)<module>()
-> def f(n):
(Pdb) break 11
Breakpoint 1 at .../pdb_no_jump.py:11

(Pdb) continue
> .../pdb_no_jump.py(11)f()
-> j = 0

(Pdb) where
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
bdb.py(379)run()
-> exec cmd in globals, locals
  <string>(1)<module>()
  .../pdb_no_jump.py(21)<module>()
-> print f(5)
```



```

> .../pdb_no_jump.py(11)f()
-> j = 0

(Pdb) up
> .../pdb_no_jump.py(21)<module>()
-> print f(5)

(Pdb) jump 25
*** You can only jump within the bottom frame

(Pdb)

```

重启程序

调试工具到达程序末尾时，它会自动重新启动，不过也可以显式地重启而不用退出调试工具，也就不会丢失当前的断点或其他设置。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann. All rights reserved.
5  #
6
7  import sys
8
9  def f():
10     print 'Command-line args:', sys.argv
11     return
12
13  if __name__ == '__main__':
14     f()

```

在调试工具中运行这个程序直至结束，会打印脚本文件的名字，因为命令行上没有给出任何其他参数。

```

$ python -m pdb pdb_run.py

> .../pdb_run.py(7)<module>()
-> import sys
(Pdb) continue

Command-line args: ['pdb_run.py']
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)

```

还可以使用 `run` 重启程序。传入 `run` 的参数将由 `shlex` 解析，并传递到程序，就好像它们是

命令行参数一样，所以程序可以用不同的设置重启。

```
(Pdb) run a b c "this is a long value"
Restarting pdb_run.py with arguments:
      a b c this is a long value
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb) continue
Command-line args: ['pdb_run.py', 'a', 'b', 'c', 'this is a long value']
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

还可以在处理中的任何其他位置使用 `run` 重启程序。

```
$ python -m pdb pdb_run.py

> .../pdb_run.py(7)<module>()
-> import sys
(Pdb) break 10
Breakpoint 1 at .../pdb_run.py:10

(Pdb) continue
> .../pdb_run.py(10)f()
-> print 'Command-line args:', sys.argv

(Pdb) run one two three
Restarting pdb_run.py with arguments:
      one two three
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

16.6.5 用别名定制调试工具

可以使用 `alias` 定义一个快捷方式，从而无须反复键入复杂的命令。别名扩展应用于每个命令的第一个词。别名的体可以由可在调试工具提示窗口合法键入的任何命令组成，包括其他调试工具命令和纯 Python 表达式。别名定义中允许递归，所以一个别名甚至可以调用另一个别名。

```
$ python -m pdb pdb_function_arguments.py

> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) break 10
Breakpoint 1 at .../pdb_function_arguments.py:10
```

```
(Pdb) continue
> .../pdb_function_arguments.py(10)recursive_function()
-> if n > 0:
```

```
(Pdb) pp locals().keys()
['output', 'n']
```

```
(Pdb) alias p1 pp locals().keys()
```

```
(Pdb) p1
['output', 'n']
```

运行 `alias` 而不带任何参数时，会显示已定义的别名的一个列表。如果给出一个参数，则认为这是别名的名，将打印这个别名的定义。

```
(Pdb) alias
p1 = pp locals().keys()
```

```
(Pdb) alias p1
p1 = pp locals().keys()
(Pdb)
```

`alias` 的参数使用 `%n` 来引用，其中 `n` 会替换为一个指示参数位置的数，从 1 开始。要利用所有参数，可以使用 `%*`。

```
$ python -m pdb pdb_function_arguments.py
```

```
> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) alias ph !help(%1)
```

```
(Pdb) ph locals
Help on built-in function locals in module __builtin__:
```

```
locals(...)
    locals() -> dictionary
```

Update and return a dictionary containing the current scope's local variables.

可以用 `unalias` 删除一个别名的定义。

```
(Pdb) unalias ph
```

```
(Pdb) ph locals
*** SyntaxError: invalid syntax (<stdin>, line 1)
```

```
(Pdb)
```

16.6.6 保存配置设置

调试一个程序需要做大量重复：运行代码、观察输出、调整代码或输入，然后再次运行。pdb 努力减少所需的重复，来控制调试体验，使你能集中精力考虑代码而不是调试工具。为了有助于减少向调试工具发出相同命令的次数，pdb 可以从启动时解释的文本文件读取已保存的配置。

首先读取文件 `~/.pdbrc`，得到所有调试会话的全局个人首选项。然后从当前工作目录读取 `./pdbrc`，设置特定项目的局部首选项。

```
$ cat ~/.pdbrc

# Show python help
alias ph !help(%1)
# Overridden alias
alias redefined p 'home definition'

$ cat ./pdbrc
# Breakpoints
break 10
# Overridden alias
alias redefined p 'local definition'

$ python -m pdb pdb_function_arguments.py

Breakpoint 1 at .../pdb_function_arguments.py:10
> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) alias
ph = !help(%1)
redefined = p 'local definition'

(Pdb) break
Num Type      Disp Enb   Where
1  breakpoint keep yes    at .../pdb_function_arguments.py:10

(Pdb)
```

能在调试工具提示窗口键入的所有配置命令都可以保存到某个启动文件中，不过大多数控制执行的命令（`continue`、`jump` 等等）除外。`run` 是个例外，这说明可以在 `./pdbrc` 中设置一个调试会话的命令行参数，从而在多次运行时保持一致。

参见：

`pdb` (<http://docs.python.org/library/pdb.html>) 这个模块的标准库文档。

`readline` (14.4 节) 交互式提示语编辑库。

`cmd` (14.6 节) 构建交互式程序。

shlex (14.7 节) shell 命令行解析。

16.7 trace——执行程序流

作用：监视程序运行时执行了哪些语句和函数，来生成覆盖和调用图信息。

Python 版本：2.3 及以后版本

trace 模块对于了解程序以何种方式运行很有用。它会监视所执行的语句，生成覆盖报告，并有助于研究相互调用的函数之间的关系。

16.7.1 示例程序

本节余下的例子都会使用这个程序。它导入另一个名为 `recurse` 的模块，然后运行其中的一个函数。

```
from recurse import recurse

def main():
    print 'This is the main program.'
    recurse(2)
    return

if __name__ == '__main__':
    main()

recurse() 函数会调用其自身，直至 level 参数达到 0。

def recurse(level):
    print 'recurse(%s)' % level
    if level:
        recurse(level-1)
    return

def not_called():
    print 'This function is never called.'
```

16.7.2 跟踪执行

可以从命令行直接使用 `trace`，这很容易。给定 `--trace` 选项时，会打印程序运行时执行的语句。

```
$ python -m trace --trace trace_example/main.py

--- modulename: threading, funcname: settrace
threading.py(89): _trace_hook = func
--- modulename: trace, funcname: <module>
<string>(1): --- modulename: trace, funcname: <module>
main.py(7): """
main.py(12): from recurse import recurse
```

```

--- modulename: recurse, funcname: <module>
recurse.py(7): """
recurse.py(12): def recurse(level):
recurse.py(18): def not_called():
main.py(14): def main():
main.py(19): if __name__ == '__main__':
main.py(20):     main()
--- modulename: trace, funcname: main
main.py(15):     print 'This is the main program.'
This is the main program.
main.py(16):     recurse(2)
--- modulename: recurse, funcname: recurse
recurse.py(13):     print 'recurse(%s)' % level
recurse(2)
recurse.py(14):     if level:
recurse.py(15):         recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13):     print 'recurse(%s)' % level
recurse(1)
recurse.py(14):     if level:
recurse.py(15):         recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13):     print 'recurse(%s)' % level
recurse(0)
recurse.py(14):     if level:
recurse.py(16):     return
recurse.py(16):     return
recurse.py(16):     return
main.py(17):     return

```

输出的第一部分显示了 trace 完成的建立操作。输出的余下部分显示了每个函数的入口，包括函数所在的模块，然后是执行时源文件的代码行。不出所料，根据 main() 中调用的方式，会 3 次进入 recurse() 函数。

16.7.3 代码覆盖

给定 --count 选项时，从命令行运行 trace 会生成代码覆盖报告信息，详细列出运行了哪些代码行，哪些行被跳过。由于复杂的程序通常由多个文件组成，所以会为各个文件分别生成一个单独的覆盖报告。默认情况下，覆盖报告文件会写至模块所在的目录，根据模块来命名，不过扩展名是 .cover 而不是 .py。

```
$ python -m trace --count trace_example/main.py
```

```

This is the main program.
recurse(2)
recurse(1)

```

```
recurse(0)
```

这里会生成两个输出文件。以下是 `trace_example/main.cover`。

```
1: from recurse import recurse

1: def main():
1:     print 'This is the main program.'
1:     recurse(2)
1:     return

1: if __name__ == '__main__':
1:     main()
```

下面是 `trace_example/recurse.cover`。

```
1: def recurse(level):
3:     print 'recurse(%s)' % level
3:     if level:
2:         recurse(level-1)
3:     return

1: def not_called():
    print 'This function is never called.'
```

注意：尽管代码行 `def recurse(level):` 计数为 1，但这并不表示这个函数只运行了一次。它表示只执行了一次函数定义。

还可以把这个程序运行多次，提供不同的选项，来保存覆盖数据并生成一个合并的报告。

```
$ python -m trace --coverdir coverdir1 --count --file coverdir1/cove\
rage_report.dat trace_example/main.py
Skipping counts file 'coverdir1/coverage_report.dat': [Errno 2] No suc
h file or directory: 'coverdir1/coverage_report.dat'
This is the main program.
recurse(2)
recurse(1)
recurse(0)
```

```
$ python -m trace --coverdir coverdir1 --count --file coverdir1/cove\
rage_report.dat trace_example/main.py
```

```
This is the main program.
recurse(2)
recurse(1)
recurse(0)
```

```
$ python -m trace --coverdir coverdir1 --count --file coverdir1/cove\
rage_report.dat trace_example/main.py
```



```

This is the main program.
recurse(2)
recurse(1)
recurse(0)

```

一旦覆盖信息记录到 .cover 文件, 要生成报告, 可以使用 --report 选项。

```

$ python -m trace --coverdir coverdir1 --report --summary --missing \
--file coverdir1/coverage_report.dat trace_example/main.py

```

```

lines   cov%   module      (path)
599      0%    threading   (/Library/Frameworks/Python.framework/Versi
ons/2.7/lib/python2.7/threading.py)
8       100%   trace_example.main (trace_example/main.py)
8       87%   trace_example.recurse (trace_example/recurse.py)

```

由于这个程序运行了3次, 覆盖报告显示, 值比第一个报告中大3倍。--summary 选项会在输出中增加百分比覆盖信息。recurse 模块只覆盖了87%。查看 recurse 的覆盖文件可以看到, not_called() 的体实际上从未运行, 可由 >>>>> 前缀指示。

```

3: def recurse(level):
9:     print 'recurse(%s)' % level
9:     if level:
6:         recurse(level-1)
9:     return

3: def not_called():
>>>>> print 'This function is never called.'

```

16.7.4 调用关系

除了覆盖信息外, trace 还会收集和报告相互调用的函数之间的关系。

要得到所调用的函数的一个简单列表, 可以使用 --listfuncs。

```

$ python -m trace --listfuncs trace_example/main.py

```

```

This is the main program.
recurse(2)
recurse(1)
recurse(0)

```

```

functions called:
filename: /Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/threading.py, modulename: threading, funcname: settrace
filename: <string>, modulename: <string>, funcname: <module>
filename: trace_example/main.py, modulename: main, funcname: <module>
filename: trace_example/main.py, modulename: main, funcname: main
filename: trace_example/recurse.py, modulename: recurse, funcname: <mo
dule>

```

```
filename: trace_example/recurse.py, module: recurse, funcname: recurse
```

要得到调用者的更多详细信息，可以使用 `--trackcalls`。

```
$ python -m trace --listfuncs --trackcalls trace_example/main.py
```

```
This is the main program.
```

```
recurse(2)
```

```
recurse(1)
```

```
recurse(0)
```

```
calling relationships:
```

```
*** /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/tr  
ace.py ***
```

```
--> /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/  
threading.py
```

```
    trace.Trace.run -> threading.settrace
```

```
--> <string>
```

```
    trace.Trace.run -> <string>.<module>
```

```
*** <string> ***
```

```
--> trace_example/main.py
```

```
    <string>.<module> -> main.<module>
```

```
*** trace_example/main.py ***
```

```
    main.<module> -> main.main
```

```
--> trace_example/recurse.py
```

```
    main.<module> -> recurse.<module>
```

```
    main.main -> recurse.recurse
```

```
*** trace_example/recurse.py ***
```

```
    recurse.recurse -> recurse.recurse
```

16.7.5 编程接口

要想更多地控制 `trace` 接口，可以在程序中调用并使用一个 `Trace` 对象。`Trace` 支持运行一个函数或执行一个要跟踪的 Python 命令之前先建立固件和其他依赖对象。

```
import trace
from trace_example.recurse import recurse
```

```
tracer = trace.Trace(count=False, trace=True)
tracer.run('recurse(2)')
```

由于这个例子只跟踪到 `recurse()` 函数，所以输出中不包含 `main.py` 的任何信息。

```
$ python trace_run.py
```

```

--- modulename: threading, funcname: settrace
threading.py(89):      _trace_hook = func
--- modulename: trace_run, funcname: <module>
<string>(1):      --- modulename: recurse, funcname: recurse
recurse.py(13):      print 'recurse(%s)' % level
recurse(2)
recurse.py(14):      if level:
recurse.py(15):      'recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13):      print 'recurse(%s)' % level
recurse(1)
recurse.py(14):      if level:
recurse.py(15):      recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13):      print 'recurse(%s)' % level
recurse(0)
recurse.py(14):      if level:
recurse.py(16):      return
recurse.py(16):      return
recurse.py(16):      return

```

用 `runfunc()` 方法也可以生成同样的输出。

```

import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.runfunc(recurse, 2)

```

`runfunc()` 接受任意的位置和关键字参数，由 `tracer` 调用时这些参数会传递到函数。

```
$ python trace_runfunc.py
```

```

--- modulename: recurse, funcname: recurse
recurse.py(13):      print 'recurse(%s)' % level
recurse(2)
recurse.py(14):      if level:
recurse.py(15):      recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13):      print 'recurse(%s)' % level
recurse(1)
recurse.py(14):      if level:
recurse.py(15):      recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13):      print 'recurse(%s)' % level
recurse(0)
recurse.py(14):      if level:
recurse.py(16):      return
recurse.py(16):      return
recurse.py(16):      return

```



16.7.6 保存结果数据

类似于命令行接口，还可以记录统计和覆盖信息。这些数据必须使用 Trace 对象的 CoverageResults 实例显式保存。

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True, trace=False)
tracer.runfunc(recurse, 2)

results = tracer.results()
results.write_results(covdir='coverdir2')
```

这个例子将覆盖结果保存到目录 coverdir2。

```
$ python trace_CoverageResults.py

recurse(2)
recurse(1)
recurse(0)

$ find coverdir2

coverdir2
coverdir2/trace_example.recurse.cover
```

输出文件包含以下内容。

```
#!/usr/bin/env python
# encoding: utf-8
#
# Copyright (c) 2008 Doug Hellmann All rights reserved.
#
"""

__version__ = "$Id$"
#end_pymotw_header

>>>>> def recurse(level):
3:     print 'recurse(%s)' % level
3:     if level:
2:         recurse(level-1)
3:     return

>>>>> def not_called():
>>>>>     print 'This function is never called.'
```

要保存统计数据来生成报告，可以对 Trace 使用 infile 和 outfile 参数。



```

import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True,
                     trace=False,
                     outfile='trace_report.dat')
tracer.runfunc(recurse, 2)

report_tracer = trace.Trace(count=False,
                             trace=False,
                             infile='trace_report.dat')
results = tracer.results()
results.write_results(summary=True, coverdir='/tmp')

```

将一个文件名传至 `infile` 来读取先前存储的数据，将一个文件名传至 `outfile` 可以在跟踪之后将新结果写入该文件。如果 `infile` 和 `outfile` 相同，其效果就是用累积的数据更新文件。

```
$ python trace_report.py
```

```

recurse(2)
recurse(1)
recurse(0)
lines   cov%   module   (path)
   7     57%   trace_example.recurse   (.../recurse.py)

```

16.7.7 选项

`Trace` 的构造函数有多个可选的参数，用来控制运行时行为。

`count` 布尔值 打开行号统计。默认为 `True`。

`countfuncs` 布尔值 打开运行期间调用的函数列表。默认为 `False`。

`countcallers` 布尔值 打开对调用者和被调用者的跟踪。默认为 `False`。

`ignoremods` 序列 跟踪覆盖情况时要忽略的模块或包列表。默认为一个空元组。

`ignoredirs` 序列 这个目录列表包含要忽略的模块或包。默认为一个空元组。

`infile` 包含缓存计数值的文件的文件名。默认为 `None`。

`outfile` 这个文件用来存储缓存计数文件。默认为 `None`，数据未存储。

参见：

`trace` (<http://docs.python.org/lib/module-trace.html>) 这个模块的标准库文档。

17.2.7 节 `sys` 模块包含一些工具，可以在运行时为解释器添加一个定制跟踪函数。

`coverage.py` (<http://nedbatchelder.com/code/modules/coverage.html>) Ned Batchelder 编写的覆盖模块。

`figleaf` (<http://darcs.idyll.org/t/projects/figleaf/doc/>) Titus Brown 编写的覆盖应用。

16.8 profile 和 pstats——性能分析

作用：Python 程序的性能分析。

Python 版本：1.4 及以后版本

profile 和 cProfile 模块提供了一些 API，用来收集和分析 Python 源代码消耗处理器资源的有关统计信息。

注意：本节的输出报告已经重新调整了格式，以适合篇幅显示。以反斜线 (\) 结尾的行表示未完，下一行仍继续。

16.8.1 运行性能分析工具

profile 模块中最基本的起点是 run()。它取一个字符串语句为参数，创建一个报告，指出运行这个语句时执行不同代码行所花费的时间。

```
import profile

def fib(n):
    # from literateprograms.org
    # http://bit.ly/h1OQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = [ ]
    if n > 0:
        seq.extend(fib_seq(n-1))
    seq.append(fib(n))
    return seq

profile.run('print fib_seq(20); print')
```

这是一个递归版本的 Fibonacci 序列计算器，对于展示 profile 尤其有用，因为其性能可以显著改善。标准报告格式会显示一个总结，然后是给出执行的各个函数的详细信息。

```
$ python profile_fibonacci_raw.py
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181, 6765]
```

```
57356 function calls (66 primitive calls) in 0.746 CPU seconds
```

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
21	0.000	0.000	0.000	0.000	:0(append)
20	0.000	0.000	0.000	0.000	:0(extend)
1	0.001	0.001	0.001	0.001	:0(setprofile)
1	0.000	0.000	0.744	0.744	<string>:1(<module>)
1	0.000	0.000	0.746	0.746	profile:0(\
					print fib_seq(20);print)
0	0.000		0.000		profile:0(profiler)
57291/21	0.743	0.000	0.743	0.035	profile_fibonacci_raw.py\ :10(fib)
21/1	0.001	0.000	0.744	0.744	profile_fibonacci_raw.py\ :20(fib_seq)

原始版本有 57 356 个不同的函数调用，运行时间为 3/4 秒。实际上这里只有 66 个基本调用，这个事实说明这 57 000 多个调用中大部分都是递归调用。列表中所用时间的详细信息按函数分解，显示了调用数、函数花费的总时间，每个调用花费的时间 (totttime/ncalls)，一个函数花费的累积时间，以及累积时间与基本调用之比。

并不奇怪，这里大部分时间都花费在反复调用 fib() 上。添加一个 memoize 修饰符可以减少递归调用数，会对这个函数的性能有很大影响。

```
import profile

class memoize:
    # from Avinash Vora's memoize decorator
    # http://bit.ly/fGzfr7
    def __init__(self, function):
        self.function = function
        self.memoized = {}

    def __call__(self, *args):
        try:
            return self.memoized[args]
        except KeyError:
            self.memoized[args] = self.function(*args)
            return self.memoized[args]

@memoize
def fib(n):
    # from literateprograms.org
    # http://bit.ly/h1OQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
```

```

        return fib(n-1) + fib(n-2)

def fib_seq(n):
    seq = [ ]
    if n > 0:
        seq.extend(fib_seq(n-1))
        seq.append(fib(n))
    return seq

if __name__ == '__main__':
    profile.run('print fib_seq(20); print')

```

通过记住各层的 Fibonacci 值，大多数调用都可以避免，将运行的调用减至 145 个，这只需要 0.003 秒。fib() 的 ncalls 数显示出它完全没有递归。

```
$ python profile_fibonacci_memoized.py
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765]
```

```
145 function calls (87 primitive calls) in 0.003 CPU seconds
```

```
Ordered by: standard name
```

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
21	0.000	0.000	0.000	0.000	:0(append)
20	0.000	0.000	0.000	0.000	:0(extend)
1	0.001	0.001	0.001	0.001	:0(setprofile)
1	0.000	0.000	0.002	0.002	<string>:1(<module>)
1	0.000	0.000	0.003	0.003	profile:0(\nprint fib_seq(20); print)
0	0.000		0.000		profile:0(profiler)
59/21	0.001	0.000	0.001	0.000	profile_fibonacci_\nmemoized.py:17(__call__)
21	0.000	0.000	0.001	0.000	profile_fibonacci_\nmemoized.py:24(fib)
21/1	0.001	0.000	0.002	0.002	profile_fibonacci_\nmemoized.py:35(fib_seq)

16.8.2 在上下文中运行

有时，不用为 run() 构造复杂表达式，更容易的做法是构建一个简单的表达式，并使用 runctx() 通过一个上下文为它传递参数。

```

import profile
from profile_fibonacci_memoized import fib, fib_seq

if __name__ == '__main__':
    profile.runctx('print fib_seq(n); print', globals(), {'n':20})

```


在这个例子中，n 的值通过局部变量上下文传递，而不是直接嵌入到传至 `runctx()` 的语句中。

```
$ python profile_runctx.py
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181, 6765]
```

```
145 function calls (87 primitive calls) in 0.003 CPU seconds
```

```
Ordered by: standard name
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
21	0.000	0.000	0.000	0.000	:0(append)
20	0.000	0.000	0.000	0.000	:0(extend)
1	0.001	0.001	0.001	0.001	:0(setprofile)
1	0.000	0.000	0.002	0.002	<string>:1(<module>)
1	0.000	0.000	0.003	0.003	profile:0(\
					print fib_seq(n); print)
0	0.000		0.000		profile:0(profile)
59/21	0.001	0.000	0.001	0.000	profile_fibonacci_\
					memoized.py:17(__call__)
21	0.000	0.000	0.001	0.000	profile_fibonacci_\
					memoized.py:24(fib)
21/1	0.001	0.000	0.002	0.002	profile_fibonacci_\
					memoized.py:35(fib_seq)

16.8.3 pstats: 保存和处理统计信息

`profile` 函数创建的标准报告不太灵活。不过，可以保存 `run()` 和 `runctx()` 的原始性能数据并用 `pstats.Stats` 类单独处理，来生成定制报告。

下面的例子多次迭代运行同一个测试，并合并结果。

```
import cProfile as profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Create 5 set of stats
filenames = []
for i in range(5):
    filename = 'profile_stats_%d.stats' % i
    profile.run('print %d, fib_seq(20)' % i, filename)

# Read all 5 stats files into a single object
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_%d.stats' % i)

# Clean up filenames for the report
```

```

stats.strip_dirs()

# Sort the statistics by the cumulative time spent in the function
stats.sort_stats('cumulative')

stats.print_stats()

```

输出报告按函数所花费累积时间的降序排序，另外打印的文件名中去掉了目录名，以节省页面上的水平空间。

```
$ python profile_stats.py
```

```

0 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
  987, 1597, 2584, 4181, 6765]
1 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
  987, 1597, 2584, 4181, 6765]
2 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
  987, 1597, 2584, 4181, 6765]
3 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
  987, 1597, 2584, 4181, 6765]
4 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
  987, 1597, 2584, 4181, 6765]

```

```

Sun Aug 31 11:29:36 2008    profile_stats_0.stats
Sun Aug 31 11:29:36 2008    profile_stats_1.stats
Sun Aug 31 11:29:36 2008    profile_stats_2.stats
Sun Aug 31 11:29:36 2008    profile_stats_3.stats
Sun Aug 31 11:29:36 2008    profile_stats_4.stats

```

489 function calls (351 primitive calls) in 0.008 CPU seconds

Ordered by: cumulative time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
5	0.000	0.000	0.007	0.001	<string>:1(<module>)
105/5	0.004	0.000	0.007	0.001	profile_fibonacci\memoized.py:36(fib_seq)
1	0.000	0.000	0.003	0.003	profile:0(print 0, \fib_seq(20))
143/105	0.001	0.000	0.002	0.000	profile_fibonacci\memoized.py:19(__call__)
1	0.000	0.000	0.001	0.001	profile:0(print 4, \fib_seq(20))
1	0.000	0.000	0.001	0.001	profile:0(print 1, \fib_seq(20))
1	0.000	0.000	0.001	0.001	profile:0(print 2, \fib_seq(20))
1	0.000	0.000	0.001	0.001	profile:0(print 3, \fib_seq(20))

21	0.000	0.000	0.001	0.000	profile_fibonacci_\nmemoized.py:26(fib)
100	0.001	0.000	0.001	0.000	:0(extend)
105	0.001	0.000	0.001	0.000	:0(append)
5	0.001	0.000	0.001	0.000	:0(setprofile)
0	0.000		0.000		profile:0(profiler)

16.8.4 限制报告内容

可以由函数限制输出。下面这个版本只显示 fib() 和 fib_seq() 性能的有关信息，这里使用一个正则表达式来匹配所需的 filename:lineno(function) 值。

```
import profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Read all 5 stats files into a single object
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_%d.stats' % i)
stats.strip_dirs()
stats.sort_stats('cumulative')

# limit output to lines with "(fib" in them
stats.print_stats('\(fib')
```

正则表达式包含一个字面量（左括号 [()），来匹配位置值的函数名部分。

```
$ python profile_stats_restricted.py
```

```
Sun Aug 31 11:29:36 2008    profile_stats_0.stats
Sun Aug 31 11:29:36 2008    profile_stats_1.stats
Sun Aug 31 11:29:36 2008    profile_stats_2.stats
Sun Aug 31 11:29:36 2008    profile_stats_3.stats
Sun Aug 31 11:29:36 2008    profile_stats_4.stats
```

```
489 function calls (351 primitive calls) in 0.008 CPU seconds
```

```
Ordered by: cumulative time
```

```
List reduced from 13 to 2 due to restriction <'\\(fib'>
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
105/5	0.004	0.000	0.007	0.001	profile_fibonacci_\nmemoized.py:36(fib_seq)
21	0.000	0.000	0.001	0.000	profile_fibonacci_\nmemoized.py:26(fib)

16.8.5 调用图

Stats 还包括一些方法来打印函数的调用者和被调用者。

```
import cProfile as profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq
# Read all 5 stats files into a single object
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_%d.stats' % i)
stats.strip_dirs()
stats.sort_stats('cumulative')

print 'INCOMING CALLERS:'
stats.print_callers('\(fib')

print 'OUTGOING CALLEES:'
stats.print_callees('\(fib')
```

`print_callers()` 和 `print_callees()` 的参数与 `print_stats()` 的限制参数类似。输出会显示调用者、被调用者、调用数以及累积时间。

```
$ python profile_stats_callers.py
```

INCOMING CALLERS:

Ordered by: cumulative time

List reduced from 7 to 2 due to restriction <'\\(fib'>

Function	was called by...	ncalls	tottime	cumtime
profile_fibonacci_memoized.py:35(fib_seq) <-		5	0.000	0.001\
<string>:1(<module>)		100/5	0.000	0.001\
profile_fibonacci_memoized.py:35(fib_seq)				
profile_fibonacci_memoized.py:24(fib) <-		21	0.000	0.000\
profile_fibonacci_memoized.py:17(__call__)				

OUTGOING CALLEES:

Ordered by: cumulative time

List reduced from 7 to 2 due to restriction <'\\(fib'>

Function	called...	ncalls	tottime	cumtime
profile_fibonacci_memoized.py:35(fib_seq) ->		105	0.000	0.000\
profile_fibonacci_memoized.py:17(__call__)		100/5	0.000	0.001\

```

profile_fibonacci_memoized.py:35(fib_seq)          105  0.000  0.000\
(method 'append' of 'list' objects)               100  0.000  0.000\
(method 'extend' of 'list' objects)
profile_fibonacci_memoized.py:24(fib)      ->    38  0.000  0.000\
profile_fibonacci_memoized.py:17(__call__)

```

参见:

profile and cProfile (<http://docs.python.org/lib/module-profile.html>) 这个模块的标准库文档。

pstats (<http://docs.python.org/lib/profile-stats.html>) pstats 的标准库文档。

Gprof2Dot (<http://code.google.com/p/jrfonseca/wiki/Gprof2Dot>) 性能输出数据的可视化工具。

Fibonacci numbers (Python)—LiteratePrograms ([http://en.literateprograms.org/Fibonacci_numbers_\(Python\)](http://en.literateprograms.org/Fibonacci_numbers_(Python))) 用 Python 编写的一个 Fibonacci 序列生成器实现。

Python Decorators: Syntactic Sugar | avinash.vora(<http://avinashv.net/2008/04/python-decorators-syntactic-sugar/>) 另一个用 Python 完成的缓存优化的 Fibonacci 序列生成器。

16.9 timeit——测量小段 Python 代码的执行时间

作用: 测量小段 Python 代码的执行时间。

Python 版本: 2.3 及以后版本

timeit 模块提供了一个简单的接口来确定小段 Python 代码的执行时间。它使用一个平台特定的时间函数, 尽可能提供最准确的时间计算, 并减少反复执行代码时启动或关闭对时间计算的影响。

16.9.1 模块内容

timeit 定义了一个公共类 Timer。Timer 的构造函数有两个参数, 一个是要测量时间的语句, 另一个是“建立”语句(例如, 用来初始化变量)。Python 语句应当是字符串, 可以包含嵌入的换行符。

timeit() 方法会运行一次建立语句, 然后反复执行主语句, 并返回过去了多少时间。timeit() 的参数控制着要运行多少次语句; 默认为 1 000 000。

16.9.2 基本示例

为了展示如何使用 Timer 的各个参数, 下面给出一个简单的例子, 执行各个语句时会打印一个标识值。

```

import timeit

# using setitem
t = timeit.Timer("print 'main statement'", "print 'setup'")

```

```

print 'TIMEIT:'
print t.timeit(2)

print 'REPEAT:'
print t.repeat(3, 2)

```

运行时，输出如下：

```
$ python timeit_example.py
```

```

TIMEIT:
setup
main statement
main statement
2.86102294922e-06
REPEAT:
setup
main statement
main statement
setup
main statement
main statement
setup
main statement
main statement
[9.5367431640625e-07, 1.9073486328125e-06, 2.1457672119140625e-06]

```

`timeit()` 运行一次建立语句，然后调用 `count` 次主语句。它返回一个浮点值，表示运行主语句花费的累积时间。

使用 `repeat()` 时，它会调用多次 `timeit()`（在这里是 3 次），所有响应都返回到一个列表中。

16.9.3 值存储在字典中

这个例子更为复杂，它比较了使用不同方法用大量值填充一个字典所需的时间。首先，需要一些常量来配置 `Timer`。`setup_statement` 变量初始化一个元组列表，这些元组中包含主语句用来构建字典的字符串和整数，用字符串作为键，并存储整数作为关联值。

```

import timeit
import sys

# A few constants
range_size=1000
count=1000
setup_statement = "l = [ (str(x), x) for x in range(1000) ]; d = {}"

```

这里定义了一个工具函数 `show_results()`，它采用一种有用的格式打印结果。`timeit()` 方法返回反复执行这个语句所花费的时间。`show_results()` 的输出将这个时间转换为每次迭代花费的

时间，然后进一步将这个值缩减为在字典中存储一项所花费的平均时间。

```
def show_results(result):
    "Print results in terms of microseconds per pass and per item."
    global count, range_size
    per_pass = 1000000 * (result / count)
    print '%.2f usec/pass' % per_pass,
    per_item = per_pass / range_size
    print '%.2f usec/item' % per_item

print "%d items" % range_size
print "%d iterations" % count
print
```

为了建立一个基准，测试的第一个配置使用了 `__setitem__()`。所有其他版本都不会覆盖字典中已经有的值，所以这个简单版本应该是最快的。

Timer 的第一个参数是一个多行的字符串，这里保留了空白符以确保运行时能正确解析。第二个参数是一个常量，用来初始化值列表和字典。

```
# Using __setitem__ without checking for existing values first
print '__setitem__:',
t = timeit.Timer("""
for s, i in l:
    d[s] = i
""",
    .
    setup_statement)
show_results(t.timeit(number=count))
```

下一个版本使用 `setdefault()` 来确保字典中已有的值不会被覆盖。

```
# Using setdefault
print 'setdefault :',
t = timeit.Timer("""
for s, i in l:
    d.setdefault(s, i)
""",
    .
    setup_statement)
show_results(t.timeit(number=count))
```

要避免覆盖现有的值，另一种方法是使用 `has_key()` 显式地检查字典的内容。

```
# Using has_key
print 'has_key :',
t = timeit.Timer("""
for s, i in l:
    if not d.has_key(s):
        d[s] = i
""",
    .
    setup_statement)
show_results(t.timeit(number=count))
```



如果查找现有值时产生一个 `KeyError` 异常，这个方法才会增加值。

```
# Using exceptions
print 'KeyError :',
t = timeit.Timer("""
for s, i in l:
    try:
        existing = d[s]
    except KeyError:
        d[s] = i
""",
setup_statement)
show_results(t.timeit(number=count))
```

最后一个方法是一种相当新的格式，这里使用 `"in"` 来确定字典是否有某个特定的键。

```
# Using "in"
print '"not in" :',
t = timeit.Timer("""
for s, i in l:
    if s not in d:
        d[s] = i
""",
setup_statement)
show_results(t.timeit(number=count))
```

运行时，脚本会生成以下输出。

```
$ python timeit_dictionary.py

1000 items
1000 iterations

__setitem__: 131.44 usec/pass 0.13 usec/item
setdefault : 282.94 usec/pass 0.28 usec/item
has_key     : 202.40 usec/pass 0.20 usec/item
KeyError    : 142.50 usec/pass 0.14 usec/item
"not in"    : 104.60 usec/pass 0.10 usec/item
```

这是在运行 Python 2.7 的 MacBook Pro 上得到的执行时间，取决于系统上运行的其他程序，这些结果可能会有所不同。可以尝试不同的 `range_size` 和 `count` 变量，因为不同的组合可能会生成不同的结果。

16.9.4 从命令行执行

除了编程接口外，`timeit` 还提供了一个命令行接口来测试模块，而不需要自动化测试 (instrumentation)。

要运行模块，可以对 Python 解释器使用 `-m` 选项来查找模块，并把它作为主程序。


```
$ python -m timeit
```

例如，使用这个命令来获得帮助。

```
$ python -m timeit -h
```

```
Tool for measuring execution time of small code snippets.
```

```
This module avoids a number of common traps for measuring execution
times. See also Tim Peters' introduction to the Algorithms chapter in
the Python Cookbook, published by O'Reilly.
```

```
...
```

命令行上的 `statement` 参数与 `Timer` 的参数稍有不同。并不是传入一个长字符串，而是要将每行指令作为一个单独的命令行参数。如果需要缩进行（如在一个循环中），可以用引号包围代码行从而在字符串中嵌入空格。

```
$ python -m timeit -s "d={}" "for i in range(1000):" " d[str(i)] = i"
```

```
1000 loops, best of 3: 559 usec per loop
```

还可以用更复杂的代码定义一个函数，然后从命令行调用这个函数。

```
def test_setitem(range_size=1000):
    l = [ (str(x), x) for x in range(range_size) ]
    d = {}
    for s, i in l:
        d[s] = i
```

要运行测试，可以传入导入模块并运行测试函数的代码。

```
$ python -m timeit "import timeit_setitem; timeit_setitem.test_
setitem()"
```

```
1000 loops, best of 3: 804 usec per loop
```

参见：

`timeit` (<http://docs.python.org/lib/module-timeit.html>) 这个模块的标准库文档。

`profile` (16.8 节) `profile` 模块对于性能分析也很有用。

16.10 compileall——字节编译源文件

作用：将源文件转换为字节编译版本。

Python 版本：1.4 及以后版本

`compileall` 模块查找 Python 源文件，并把它们编译为字节码表示，将结果保存在 `.pyc` 或 `.pyo` 文件中。

16.10.1 编译一个目录

`compile_dir()` 用于递归地扫描一个目录，并对其中的文件完成字节编译。

```
import compileall
```

```
compileall.compile_dir('examples')
```

默认情况下，所有子目录都会得到扫描，直至深度达到 10。

```
$ python compileall_compile_dir.py
```

```
Listing examples ...
Compiling examples/a.py ...
Listing examples/subdir ...
Compiling examples/subdir/b.py ...
```

要筛除目录，可以使用 `rx` 参数提供一个正则表达式来匹配要排除的目录名。

```
import compileall
import re
```

```
compileall.compile_dir('examples',
    rx=re.compile(r'/subdir'))
```

这个版本会排除 `subdir` 子目录中的文件。

```
$ python compileall_exclude_dirs.py
```

```
Listing examples ...
Compiling examples/a.py ...
Listing examples/subdir ...
```

`maxlevels` 参数控制递归深度。例如，要完全避免递归，可以传入 0。

```
import compileall
import re
```

```
compileall.compile_dir('examples',
    maxlevels=0,
    rx=re.compile(r'/\.svn'))
```

这样一来，只会编译传递到 `compile_dir()` 的目录中的文件。

```
$ python compileall_recursion_depth.py
```

```
Listing examples ...
Compiling examples/a.py ...
```

16.10.2 编译 sys.path

只需一个 `compile_path()` 调用，就可以编译 `sys.path` 中找到的所有 Python 源文件。

```
import compileall
```

```
import sys

sys.path[:] = ['examples', 'notthere']
print 'sys.path =', sys.path
compileall.compile_path()
```

这个例子替换了 `sys.path` 的默认内容，以避免运行脚本时的权限错误，不过仍能很好地展示默认行为。注意 `maxlevels` 值默认为 0。

```
$ python compileall_path.py
```

```
sys.path = ['examples', 'notthere']
Listing examples ...
Compiling examples/a.py ...
Listing notthere ...
Can't list notthere
```

16.10.3 从命令行执行

也可以从命令行调用 `compileall`，从而能通过一个 `Makefile` 与一个构建系统集成。下面给出一个例子。

```
$ python -m compileall -h
```

```
option -h not recognized
usage: python compileall.py [-l] [-f] [-q] [-d destdir] [-x
regexp] [-i list] [directory|file ...]
-l: don't recurse down
-f: force rebuild even if timestamps are up-to-date
-q: quiet operation
-d destdir: purported directory name for error messages
    if no directory arguments, -l sys.path is assumed
-x regexp: skip files matching the regular expression regexp
    the regexp is searched for in the full path of the file
-i list: expand list with its content (file and directory names)
```

要重新创建之前的例子，跳过 `subdir` 目录，可以运行以下命令。

```
$ python -m compileall -x '/subdir' examples
```

```
Listing examples ...
Compiling examples/a.py ...
Listing examples/subdir ...
```

参见：

`compileall` (<http://docs.python.org/library/compileall.html>) 这个模块的标准库文档。

16.11 pycbr——类浏览器

作用：实现一个适用于源代码编辑器的 API，可以用来建立一个类浏览器。

Python 版本: 1.4 及以后版本

`pyclbr` 可以扫描 Python 源代码来查找类和独立的函数。可以使用 `tokenize` 收集类、方法和函数名及行号的有关信息, 而无须导入代码。

本节中的例子都使用以下源文件作为输入。

```
"""Example source for pyclbr.
"""

class Base(object):
    """This is the base class.
    """

    def method1(self):
        return

class Sub1(Base):
    """This is the first subclass.
    """

class Sub2(Base):
    """This is the second subclass.
    """

class Mixin:
    """A mixin class.
    """

    def method2(self):
        return

class MixinUser(Sub2, Mixin):
    """Overrides method1 and method2
    """

    def method1(self):
        return

    def method2(self):
        return

    def method3(self):
        return

def my_function():
    """Stand-alone function.
    """
    return
```



16.11.1 扫描类

pyclbr 公布了两个公共函数。第一个是 readmodule(), 它以模块名作为参数, 并返回一个字典, 将类名映射到 Class 对象, 其中包含有关类源代码的元数据。

```
import pyclbr
import os
from operator import itemgetter

def show_class(name, class_data):
    print 'Class:', name
    filename = os.path.basename(class_data.file)
    print '\tFile: {0} [{1}]'.format(filename, class_data.lineno)
    show_super_classes(name, class_data)
    show_methods(name, class_data)
    print
    return

def show_methods(class_name, class_data):
    for name, lineno in sorted(class_data.methods.items(),
                               key=itemgetter(1)):
        print '\tMethod: {0} [{1}]'.format(name, lineno)
    return

def show_super_classes(name, class_data):
    super_class_names = []
    for super_class in class_data.super:
        if super_class == 'object':
            continue
        if isinstance(super_class, basestring):
            super_class_names.append(super_class)
        else:
            super_class_names.append(super_class.name)
    if super_class_names:
        print '\tSuper classes:', super_class_names
    return

example_data = pyclbr.readmodule('pyclbr_example')

for name, class_data in sorted(example_data.items(),
                               key=lambda x:x[1].lineno):
    show_class(name, class_data)
```

类的源数据包括定义这个类的文件及所在的行号, 还包括超类的类名。类的方法保存为方法名与行号之间的一个映射。输出显示了这些类和方法 (根据它们在源文件中的行号排序)。

```
$ python pyclbr_readmodule.py
```

```
Class: Base
```

```

File: pycldr_example.py [10]
Method: method1 [14]

Class: Sub1
File: pycldr_example.py [17]
Super classes: ['Base']

Class: Sub2
File: pycldr_example.py [21]
Super classes: ['Base']

Class: Mixin
File: pycldr_example.py [25]
Method: method2 [29] _

Class: MixinUser
File: pycldr_example.py [32]
Super classes: ['Sub2', 'Mixin']
Method: method1 [36]
Method: method2 [39]
Method: method3 [42]

```

16.11.2 扫描函数

pycldr 中的另一个公共函数是 `readmodule_ex()`。它能完成 `readmodule()` 所做的全部工作，并为结果集添加了一些函数。

```

import pycldr
import os
from operator import itemgetter

example_data = pycldr.readmodule_ex('pycldr_example')

for name, data in sorted(example_data.items(), key=lambda x:x[1].
    lineno):
    if isinstance(data, pycldr.Function):
        print 'Function: {0} [{1}]'.format(name, data.lineno)

```

每个 Function 对象的属性与 Class 对象很类似。

```
$ python pycldr_readmodule_ex.py
```

```
Function: my_function [45]
```

参见：

`pycldr` (<http://docs.python.org/library/pycldr.html>) 这个模块的标准库文档。

`inspect` (18.4 节) `inspect` 模块可以发现有关类和函数的更多元数据，不过需要导入代码。

`tokenize` `tokenize` 模块将 Python 源代码解析为 token。

第 17 章

运行时特性

本章将介绍 Python 标准库中的一些运行时特性，利用这些特性，程序可以与解释器或它运行所在的环境交互。

启动时，解释器会加载 `site` 模块来配置特定于当前安装的设置。通过结合环境设置、解释器构建参数以及配置文件来构造导入路径。

`sys` 模块是标准库中最大的模块之一。它包含大量函数来访问各种解释器和系统设置，包括解释器构建设置和限制；命令行参数和程序退出码；异常处理；线程调试和控制；导入机制和导入模块；运行时控制流跟踪；以及进程的标准输入和输出流。

`sys` 的重点是解释器设置，`os` 则允许访问操作系统信息。这个模块可以用作系统调用的可移植接口，会返回正在运行的进程的有关详细信息，如进程所有者和环境变量。它还包括一些函数来处理系统和进程管理。

Python 通常用作跨平台语言，用来创建可移植程序。即使一个程序可能在任意的环境中运行，有时也有必要知道当前系统的操作系统或硬件体系结构。`platform` 提供了一些函数来获取运行时设置。

可以通过 `resource` 模块探查和修改系统资源的限制，如最大进程栈大小或打开的文件数目。它还会报告当前消耗率，从而能监视进程的资源泄漏情况。

`gc` 模块允许访问 Python 垃圾回收系统的内部状态。它包括一些有用的信息来检测和中断对象周期、打开和关闭垃圾收集器，以及调整自动触发垃圾回收清扫的阈值。

`sysconfig` 模块包含构建脚本的编译时变量。构建和打包工具可以用这个模块来动态生成路径和其他设置。

17.1 site——全站点配置

`site` 模块处理站点特定的配置，特别是导入路径。

17.1.1 导入路径

每次解释器启动时会自动导入 `site`。导入时，会用站点特定的名称扩展 `sys.path`，这个名称通过组合前缀值（`sys.prefix` 和 `sys.exec_prefix`）与一些后缀值来构造。使用的前缀值将保存在模块级变量 `PREFIXES` 中，以便以后引用。在 Windows 下，后缀值是一个空串和 `lib/site-packages`。对于类 UNIX 平台，后缀值是 `lib/python$version/site-packages`（其中 `$version` 要替换

为解释器的主版本号和次版本号，如 2.7) 和 lib/site-python。

```
import sys
import os
import platform
import site

if 'Windows' in platform.platform():
    SUFFIXES = [
        '',
        'lib/site-packages',
    ]
else:
    SUFFIXES = [
        'lib/python%s/site-packages' % sys.version[:3],
        'lib/site-python',
    ]

print 'Path prefixes:'
for p in site.PREFIXES:
    print ' ', p

for prefix in sorted(set(site.PREFIXES)):
    print
    print prefix
    for suffix in SUFFIXES:
        print
        print ' ', suffix
        path = os.path.join(prefix, suffix).rstrip(os.sep)
        print '   exists :', os.path.exists(path)
        print '   in path:', path in sys.path
```

对组合得到的各个路径进行测试，将确实存在的路径添加到 sys.path。这个输出显示了安装在 Mac OS X 系统上的一个 Python 框架版本。

```
$ python site_import_path.py

Path prefixes:
/Library/Frameworks/Python.framework/Versions/2.7
/Library/Frameworks/Python.framework/Versions/2.7

/Library/Frameworks/Python.framework/Versions/2.7

lib/python2.7/site-packages
exists : True
in path: True

lib/site-python
exists : False
```



```
in path: False
```

17.1.2 用户目录

除了全局的 `site-package` 路径, `site` 还负责向导入路径添加用户特定的位置。用户特定的路径都基于 `USER_BASE` 目录, 这通常位于当前用户拥有 (而且可写) 的部分文件系统。`USER_BASE` 目录中有一个 `site-packages` 目录, 其中包含的路径可以用 `USER_SITE` 访问。

```
import site
```

```
print 'Base:', site.USER_BASE
print 'Site:', site.USER_SITE
```

`USER_SITE` 路径名使用同样的平台特定后缀 (如前所述) 来创建。

```
$ python site_user_base.py
```

```
Base: /Users/dhellmann/.local
Site: /Users/dhellmann/.local/lib/python2.7/site-packages
```

用户基目录可以通过 `PYTHONUSERBASE` 环境变量设置, 并有平台特定的默认值 (对于 Windows 是 `~/Python$version/site-packages`, 对于非 Windows 平台为 `~/.local`)。

```
$ PYTHONUSERBASE=/tmp/$USER python site_user_base.py
```

```
Base: /tmp/dhellmann
Site: /tmp/dhellmann/lib/python2.7/site-packages
```

一些可能出现安全问题的情况下会禁用用户目录 (例如, 如果运行进程的用户或组 `id` 不同于原先启动这个进程的实际用户)。应用可以通过查看 `ENABLE_USER_SITE` 来检查这个设置。

```
import site
```

```
status = {
    None: 'Disabled for security',
    True: 'Enabled',
    False: 'Disabled by command-line option',
}
```

```
print 'Flag    : ', site.ENABLE_USER_SITE
print 'Meaning:', status[site.ENABLE_USER_SITE]
```

还可以在命令行用 `-s` 显式禁用用户目录。

```
$ python site_enable_user_site.py
```

```
Flag    : True
Meaning: Enabled
```

```
$ python -s site_enable_user_site.py
```



```
Flag      : False
Meaning: Disabled by command-line option
```

17.1.3 路径配置文件

随着路径添加到导入路径，还会在这些路径中扫描路径配置文件（path configuration file）。路径配置文件是一个纯文本文件，扩展名为 .pth。文件中的每一行可以有以下 4 种形式：

- 要添加到导入路径的另一个位置的完全或相对路径。
- 一个要执行的 Python 语句。所有这样的行都必须以一个 import 语句开头。
- 空行，这些行会被忽略。
- 以 # 开头的行，要处理为注释并被忽略。

路径配置文件可以用来扩展导入路径，来查看不能自动添加的路径。例如，Distribute 包使用 python setup.py develop 以开发模式安装一个包时会向 easy-install.pth 添加一个路径。

扩展 sys.path 的函数是公共的，可以在示例程序中用来显示路径配置文件如何工作。给定一个名为 with_modules 的目录，其中包含文件 mymodule.py，以下给出使用这个 print 语句的结果。它会显示模块如何导入。

```
import os
print 'Loaded', __name__, 'from', __file__[len(os.getcwd()):+1:]
```

下面这个脚本显示了 addsitedir() 如何扩展导入路径，使解释器能够找到所需的模块。

```
import site
import os
import sys

script_directory = os.path.dirname(__file__)
module_directory = os.path.join(script_directory, sys.argv[1])

try:
    import mymodule
except ImportError, err:
    print 'Could not import mymodule:', err

print
before_len = len(sys.path)
site.addsitedir(module_directory)
print 'New paths:'
for p in sys.path[before_len:]:
    print p.replace(os.getcwd(), '.') # shorten dirname

print
import mymodule
```

将包含模块的目录添加到 sys.path 之后，脚本可以毫无问题地导入 mymodule。

```
$ python site_addsitedir.py with_modules

Could not import mymodule: No module named mymodule

New paths:
./with_modules

Loaded mymodule from with_modules/mymodule.py
```

`addsitedir()` 完成的路径调整不只是向 `sys.path` 追加参数。如果为 `addsitedir()` 提供的目录包括与模式 `*.pth` 匹配的文件，它们会作为路径配置文件加载。例如，如果 `with_pth/pymotw.pth` 包含以下内容：

```
# Add a single subdirectory to the path.
./subdir
```

而且 `mymodule.py` 已经复制为 `with_pth/subdir/mymodule.py`，则可以通过添加 `with_pth` 作为一个站点目录将这个模块导入。即使模块不在那个目录中也是可以的，因为 `with_pth` 和 `with_pth/subdir` 都会添加到导入路径。

```
$ python site_addsitedir.py with_pth

Could not import mymodule: No module named mymodule

New paths:
./with_pth
./with_pth/subdir

Loaded mymodule from with_pth/subdir/mymodule.py
```

如果一个站点目录包含多个 `.pth` 文件，会按字母顺序进行处理。

```
$ ls -F multiple_pth

a.pth
b.pth
from_a/
from_b/

$ cat multiple_pth/a.pth

./from_a

$ cat multiple_pth/b.pth

./from_b
```

在这里，模块会在 `multiple_pth/from_a` 中找到，因为 `a.pth` 比 `b.pth` 先读取。

```
$ python site_addsitedir.py multiple_pth
```



```
Could not import mymodule: No module named mymodule
```

```
New paths:
./multiple_pth
./multiple_pth/from_a
./multiple_pth/from_b
```

```
Loaded mymodule from multiple_pth/from_a/mymodule.py
```

17.1.4 定制站点配置

site 模块还负责加载站点范围的定制设置，这个设置在 sitecustomize 模块中由本地站点所有者定义。sitecustomize 可以用来扩展导入路径，并启用覆盖、性能分析或其他开发工具。

例如，下面这个 sitecustomize.py 脚本用一个基于当前平台的目录扩展导入路径。/opt/python 中的平台特定路径会添加到导入路径，从而可以导入其中安装的所有包。如果网络中不同主机之间要通过一个共享文件系统来共享包含编译扩展模块的包，类似这样的系统会很有用。只需要在各个主机上安装 sitecustomize.py 脚本，其他包都可以从文件服务器访问。

```
print 'Loading sitecustomize.py'

import site
import platform
import os
import sys

path = os.path.join('/opt',
                    'python',
                    sys.version[:3],
                    platform.platform(),
                    )

print 'Adding new path', path

site.addsitedir(path)
```

可以用一个简单的脚本来显示 Python 开始运行你自己的代码之前会先导入 sitecustomize.py。

```
import sys

print 'Running main program'

print 'End of path:', sys.path[-1]
```

由于 sitecustomize 要用来建立全系统范围的配置，所以应当安装在默认路径上的某个位置（通常在 site-packages 目录中）。下面这个例子显式地设置了 PYTHONPATH，以确保模块可以导入。

```
$ PYTHONPATH=with_sitecustomize python with_sitecustomize/site_
sitecustomize.py
```

```

Loading sitecustomize.py
Adding new path /opt/python/2.7/Darwin-10.5.0-i386-64bit
Running main program
End of path: /opt/python/2.7/Darwin-10.5.0-i386-64bit

```

17.1.5 定制用户配置

类似于 `sitecustomize`，可以用 `usercustomize` 模块建立每次解释器启动时的用户特定设置。`usercustomize` 在 `sitecustomize` 之后加载，所以可以覆盖全站点范围的定制设置。

有些环境下，多个运行不同操作系统或不同版本的服务器会共享用户的主目录，标准用户目录机制可能不适用于用户特定的包安装。在这些情况下，可以使用一个平台特定的目录树。

```

print 'Loading usercustomize.py'

import site
import platform
import os
import sys

path = os.path.expanduser(os.path.join('~',
                                       'python',
                                       sys.version[:3],
                                       platform.platform(),
                                       ))

print 'Adding new path', path

site.addsitedir(path)

```

类似于介绍 `sitecustomize` 时所用的脚本，可以用另一个简单的脚本来显示 Python 开始运行其他代码之前会先导入 `usercustomize.py`。

```

import sys

print 'Running main program'

print 'End of path:', sys.path[-1]

```

由于 `usercustomize` 要用来建立一个用户的用户特定配置，所以应当安装在用户默认路径上的某个位置，而不是放在全站点路径上。默认的 `USER_BASE` 目录就是一个很好的位置。这个例子显式地设置了 `PYTHONPATH`，以确保模块可以导入。

```

$ PYTHONPATH=with_usercustomize python with_usercustomize/site_\
usercustomize.py

Loading usercustomize.py
Adding new path /Users/dhellmann/python/2.7/Darwin-10.5.0-i386-64bit
Running main program
End of path: /Users/dhellmann/python/2.7/Darwin-10.5.0-i386-64bit

```

禁用用户站点目录特性时，不会导入 `usercustomize`，不论它位于用户站点目录还是在其他位置。

```
$ PYTHONPATH=with_usercustomize python -s with_usercustomize/site_\
usercustomize.py
```

```
Running main program
```

```
End of path: /Library/Frameworks/Python.framework/Versions/2.7/lib/
python2.7/site-packages
```

17.1.6 禁用 site 模块

之前的 Python 版本没有添加自动导入特性，要维护与这些版本的向后兼容性，解释器还接受一个 `-S` 选项。

```
$ python -S site_import_path.py
```

```
Path prefixes:
```

```
sys.prefix      : /Library/Frameworks/Python.framework/Versions/2.7
```

```
sys.exec_prefix: /Library/Frameworks/Python.framework/Versions/2.7
```

```
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
```

```
site-packages
```

```
exists: True
```

```
in path: False
```

```
/Library/Frameworks/Python.framework/Versions/2.7/lib/site-python
```

```
exists: False
```

```
in path: False
```

参见：

`site` (<http://docs.python.org/library/site.html>) 这个模块的标准库文档。

17.2.6 节描述了 `sys` 模块中定义的导入路径如何工作。

Running code at Python startup(http://nedbatchelder.com/blog/201001/running_code_at_python_startup.html) Ned Batchelder 的一个帖子，讨论了多种方法可以让 Python 解释器启动主程序执行之前先运行定制代码。

Distribute (<http://packages.python.org/distribute>) Distribute 是基于 `setuptools` 和 `distutils` 的一个 Python 打包库。

17.2 sys——系统特定的配置

作用：提供系统特定的配置和操作。

Python 版本：1.4 及以后版本

`sys` 模块包括一组服务，可以探查或修改解释器的运行时配置以及资源，从而与当前程序之外的操作环境交互。

参见:

`sys` (<http://docs.python.org/library/sys.html>) 这个模块的标准库文档。

17.2.1 解释器设置

`sys` 包含有一些属性和函数，可以访问解释器的编译时或运行时配置设置。

构建时版本信息

构建 C 解释器所用的版本可以有多种形式。`sys.version` 是一个人类可读的串，通常包含完整的版本号，以及有关构建日期、编译器和平台的信息。`sys.hexversion` 可以更容易地检查解释器版本，因为它是一个简单的整数。使用 `hex()` 格式化时，可以清楚地看出，`sys.hexversion` 的某些部分来自更可读的 `sys.version_info` 中同样可见的版本信息（这是一个包含 5 部分的元组，只表示版本号）。

对于加入构建中的源文件，有关的更多特定信息可以从 `sys.subversion` 元组找到，其中包括签出和构建的具体分支及版本修订。当前解释器使用的 C API 版本保存在 `sys.api_version` 中。

```
import sys

print 'Version info:'
print
print 'sys.version      =', repr(sys.version)
print 'sys.version_info =', sys.version_info
print 'sys.hexversion   =', hex(sys.hexversion)
print 'sys.subversion   =', sys.subversion
print 'sys.api_version  =', sys.api_version
```

所有这些值都依赖于运行示例程序的具体解释器。

```
$ python2.6 sys_version_values.py
```

```
Version info:
```

```
sys.version      = '2.6.5 (r265:79359, Mar 24 2010, 01:32:55) \n[GCC 4
.0.1 (Apple Inc. build 5493)]'
sys.version_info = (2, 6, 5, 'final', 0)
sys.hexversion   = 0x20605f0
sys.subversion   = ('CPython', 'tags/r265', '79359')
sys.api_version  = 1013
```

```
$ python2.7 sys_version_values.py
```

```
Version info:
```

```
sys.version      = '2.7 (r27:82508, Jul  3 2010, 21:12:11) \n[GCC 4.0.
1 (Apple Inc. build 5493)]'
sys.version_info = sys.version_info(major=2, minor=7, micro=0, release
```

```

level='final', serial=0)
sys.hexversion    = 0x20700f0
sys.subversion    = ('CPython', 'tags/r27', '82508')
sys.api_version   = 1013

```

用来构建解释器的操作系统平台保存为 `sys.platform`。

```
import sys
```

```
print 'This interpreter was built for:', sys.platform
```

对于大多数 UNIX 系统，这个值由命令 `uname -s` 的输出与 `uname -r` 中版本的第一部分组合而成。对于其他操作系统，则有一个硬编码的值表。

```
$ python sys_platform.py
```

```
This interpreter was built for: darwin
```

命令行选项

CPython 解释器接受一些命令行选项来控制解释器的行为；这些选项如表 17.1 所列。

表 17.1 CPython 命令行选项标志

选 项	含 义
-B	导入时不写 .py[co] 文件
-d	调试解析器的输出
-E	忽略 PYTHON* 环境变量（如 PYTHONPATH）
-i	运行脚本后交互式检查
-O	对生成的字节码稍做优化
-OO	除了 -O 优化外，还会删除 docstring
-s	不向 sys.path 添加用户站点目录
-S	不在初始化时运行 “import site”
-t	发出警告，指出 tab 使用不一致
-tt	发出错误，指出 tab 使用不一致
-v	详细显示
-3	关于 Python 3.x 不兼容性的警告

其中一些选项可以用于程序，可以通过 `sys.flags` 来检查。

```
import sys
```

```

if sys.flags.debug:
    print 'Debugging'
if sys.flags.py3k_warning:
    print 'Warning about Python 3.x incompatibilities'
if sys.flags.division_warning:
    print 'Warning about division change'

```



```

if sys.flags.division_new:
    print 'New division behavior enabled'
if sys.flags.inspect:
    print 'Will enter interactive mode after running'
if sys.flags.optimize:
    print 'Optimizing byte-code'
if sys.flags.dont_write_bytecode:
    print 'Not writing byte-code files'
if sys.flags.no_site:
    print 'Not importing "site"'
if sys.flags.ignore_environment:
    print 'Ignoring environment'
if sys.flags.tabcheck:
    print 'Checking for mixed tabs and spaces'
if sys.flags.verbose:
    print 'Verbose mode'
if sys.flags.unicode:
    print 'Unicode'

```

可以尝试使用 `sys_flags.py` 来了解命令行选项如何映射到标志设置。

```
$ python -3 -S -E sys_flags.py
```

```

Warning about Python 3.x incompatibilities
Warning about division change
Not importing "site"
Ignoring environment
Checking for mixed tabs and spaces

```

Unicode 默认编码

要得到解释器使用的默认 Unicode 编码名，可以使用 `getdefaultencoding()`。这个值在启动时由 `site` 设置，它会调用 `sys.setdefaultencoding()`，然后从 `sys` 的命名空间将其删除，避免再次调用。

对于某些操作系统，内部编码默认设置和文件系统编码可能不同，所以要有另外一种方法获取文件系统设置。`getfilesystemencoding()` 会返回一个操作系统特定的值（而不是文件系统特定的值）。

```
import sys
```

```

print 'Default encoding      :', sys.getdefaultencoding()
print 'File system encoding  :', sys.getfilesystemencoding()

```

大多数 Unicode 专家并不建议改变全局默认编码，而是推荐让应用显式地设置 Unicode。这种方法提供了两个好处：不同的数据源有不同的 Unicode 编码，这样可以更简洁地处理，而且可以减少对应用代码中编码的假设。

```
$ python sys_unicode.py
```

```
Default encoding      : ascii
File system encoding : utf-8
```

交互式提示语

交互式解释器使用两种不同的提示语来指示默认输入级 (ps1) 和多行语句的“继续” (ps2)。这些值只用于交互式解释器。

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'...'
>>>
```

这两个提示语都可以改为一个不同的串。

```
>>> sys.ps1 = '::: '
::: sys.ps2 = '~~~ '
::: for i in range(3):
~~~     print i
~~~
0
1
2
:::
```

或者，只要一个对象可以转换为字符串（通过 `__str__`），就可以用作提示语。

```
import sys

class LineCounter(object):
    def __init__(self):
        self.count = 0
    def __str__(self):
        self.count += 1
        return '(%3d)> ' % self.count
```

LineCounter 会记录它使用了多少次，所以提示语中的数字每次都会增加。

```
$ python
```

```
Python 2.6.2 (r262:71600, Apr 16 2009, 09:17:39)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from PyMOTW.sys.sys_ps1 import LineCounter
>>> import sys
>>> sys.ps1 = LineCounter()
( 1)>
( 2)>
( 3)>
```

显示 hook

每次用户进入一个表达式时交互式解释器都会调用 `sys.displayhook`。这个表达式的结果将作为惟一的参数传至函数。

```
import sys

class ExpressionCounter(object):

    def __init__(self):
        self.count = 0
        self.previous_value = self
    def __call__(self, value):
        print
        print ' Previous:', self.previous_value
        print ' New      :', value
        print
        if value != self.previous_value:
            self.count += 1
            sys.ps1 = '(%3d)> ' % self.count
        self.previous_value = value
        sys.__displayhook__(value)

print 'installing'
sys.displayhook = ExpressionCounter()
```

默认值（保存在 `sys.__displayhook__` 中）将结果打印到标准输出（`stdout`），并把它保存到 `__builtin__` 以便以后轻松引用。

```
$ python
```

```
Python 2.6.2 (r262:71600, Apr 16 2009, 09:17:39)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import PyMOTW.sys.sys_displayhook
installing
>>> 1+2
```

```
Previous: <PyMOTW.sys.sys_displayhook.ExpressionCounter object at
0x9c5f 90>
New      : 3
```

```
3
( 1)> 'abc'
```

```
Previous: 3
New      : abc
```

```
'abc'
( 2)> 'abc'

Previous: abc
New      : abc

'abc'
( 2)> 'abc' * 3
Previous: abc
New      : abcabcab

'abcbcab'
( 3)>
```

安装位置

只要系统上有一个合理的解释器路径，就可以由 `sys.executable` 得到具体的解释器程序的路径。这对于确保使用了正确的解释器可能很有用，而且还能对基于解释器位置可能设置的路径给出线索。

`sys.prefix` 指示解释器安装的父目录。它通常包括 `bin` 和 `lib` 目录，分别存放可执行文件和已安装模块。

```
import sys

print 'Interpreter executable:', sys.executable
print 'Installation prefix   :', sys.prefix
```

这个示例输出是在 Mac 上生成的，它运行了一个从 `python.org` 安装的框架构建。

```
$ python sys_locations.py
```

```
Interpreter executable: /Library/Frameworks/Python.framework/
Versions/2.7/Resources/Python.app/Contents/MacOS/Python
Installation prefix   : /Library/Frameworks/Python.framework/
Versions/2.7
```

17.2.2 运行时环境

`sys` 提供了一些底层 API 与应用的外部系统交互，可以接受命令行参数、访问用户输入，以及向用户传递消息和状态值。

命令行参数

解释器捕获的参数会由解释器处理，不会传递到它运行的程序。其余的所有选项和参数，包括脚本名本身，都保存到 `sys.argv`，以备程序使用。

```
import sys

print 'Arguments:', sys.argv
```

在第 3 个例子中，`-u` 选项由解释器处理，不会传递到它运行的程序。

```
$ python sys_argv.py
```

```
Arguments: ['sys_argv.py']
```

```
$ python sys_argv.py -v foo blah
```

```
Arguments: ['sys_argv.py', '-v', 'foo', 'blah']
```

```
$ python -u sys_argv.py
```

```
Arguments: ['sys_argv.py']
```

参见：

`getopt` (14.1 节)、`optparse` (14.2 节) 和 `argparse` (14.3 节) 解析命令行参数的模块。

输入和输出流

遵循 UNIX 编程范式，默认地 Python 程序可以访问 3 个文件描述符。

```
import sys
```

```
print >>sys.stderr, 'STATUS: Reading from stdin'
```

```
data = sys.stdin.read()
```

```
print >>sys.stderr, 'STATUS: Writing data to stdout'
```

```
sys.stdout.write(data)
```

```
sys.stdout.flush()
```

```
print >>sys.stderr, 'STATUS: Done'
```

`stdin` 是读取输入的标准方法，通常从控制台读取，不过也可以通过管道从其他程序读取。

`stdout` 是为用户写输出的标准方法（写至控制台），或者发送到管线中的下一个程序。`stderr` 用于写警告或错误消息。

```
$ cat sys_stdio.py | python sys_stdio.py
```

```
STATUS: Reading from stdin
```

```
STATUS: Writing data to stdout
```

```
#!/usr/bin/env python
```

```
# encoding: utf-8
```

```
#
```

```
# Copyright (c) 2009 Doug Hellmann All rights reserved.
```

```
#
```

```
"""
```

```
"""
```

```
#end_pymotw_header
```

```
import sys

print >>sys.stderr, 'STATUS: Reading from stdin'

data = sys.stdin.read()

print >>sys.stderr, 'STATUS: Writing data to stdout'

sys.stdout.write(data)
sys.stdout.flush()

print >>sys.stderr, 'STATUS: Done'
STATUS: Done
```

参见:

subprocess (10.1 节)和 pipes subprocess 和 pipes 都提供了相应特性可以将程序通过管线结合在一起。

返回状态

要从一个程序返回一个退出码, 需要向 sys.exit() 传递一个整数值。

```
import sys
```

```
exit_code = int(sys.argv[1])
sys.exit(exit_code)
```

非 0 值表示程序退出时有一个错误。

```
$ python sys_exit.py 0 ; echo "Exited $?"
```

```
Exited 0
```

```
$ python sys_exit.py 1 ; echo "Exited $?"
```

```
Exited 1
```

17.2.3 内存管理和限制

sys 包含一些函数来了解和控制内存使用。

引用数

Python 使用引用计数 (reference counting) 和垃圾回收 (garbage collection) 来完成自动内存管理。一个对象的引用数降至 0 时, 它会自动标志为回收。要查看一个现有对象的引用数, 可以使用 getrefcount()。

```
import sys
```

```
one = []
```

```

print 'At start          :', sys.getrefcount(one)

two = one

print 'Second reference :', sys.getrefcount(one)

del two

print 'After del         :', sys.getrefcount(one)

```

这个数实际上比期望的计数多 1, 因为 `getrefcount()` 本身会维护对象的一个临时引用。

```
$ python sys_getrefcount.py
```

```

At start          : 2
Second reference : 3
After del         : 2

```

参见:

`gc` (17.6 节) 通过 `gc` 中提供的函数控制垃圾回收器。

对象大小

了解一个对象有多少引用可以帮助发现环或内存泄漏, 不过还不足以确定哪些对象消耗的内存最多。这需要知道对象有多大。

```

import sys

class OldStyle:
    pass

class NewStyle(object):
    pass

for obj in [ [], (), {}, 'c', 'string', 1, 2.3,
             OldStyle, OldStyle(), NewStyle, NewStyle(),
             ]:
    print '%10s : %s' % (type(obj).__name__, sys.getsizeof(obj))

```

`getsizeof()` 会报告一个对象的大小 (单位为字节)。

```
$ python sys_getsizeof.py
```

```

list : 72
tuple : 56
dict : 280
str : 38
str : 43
int : 24
float : 24

```

```

classobj : 104
instance : 72
    type : 904
NewStyle : 64

```

为定制类报告的大小不包括属性值的大小。

```

import sys

class WithoutAttributes(object):
    pass

class WithAttributes(object):
    def __init__(self):
        self.a = 'a'
        self.b = 'b'
    return

without_attrs = WithoutAttributes()
print 'WithoutAttributes:', sys.getsizeof(without_attrs)

with_attrs = WithAttributes()
print 'WithAttributes:', sys.getsizeof(with_attrs)

```

这可能会让人对消耗的内存量有一个错误的印象。

```
$ python sys_getsizeof_object.py
```

```

WithoutAttributes: 64
WithAttributes: 64

```

对于一个类所用的空间，要得到更全面的估计，模块提供了一个 `__sizeof__()` 方法来计算这个值，它会累计一个对象各个属性的大小。

```

import sys

class WithAttributes(object):
    def __init__(self):
        self.a = 'a'
        self.b = 'b'
    return
    def __sizeof__(self):
        return object.__sizeof__(self) + \
            sum(sys.getsizeof(v) for v in self.__dict__.values())

my_inst = WithAttributes()
print sys.getsizeof(my_inst)

```

这个版本将对象的基本大小加上存储在内部 `__dict__` 中的所有属性的大小来计算对象大小。

```
$ python sys_getsizeof_custom.py
```


140

递归

Python 应用中允许无限递归，这可能会引入解释器本身的栈溢出，导致崩溃。为了消除这种情况，解释器提供了一种方法，可以使用 `setrecursionlimit()` 和 `getrecursionlimit()` 控制最大递归深度。

```
import sys

print 'Initial limit:', sys.getrecursionlimit()

sys.setrecursionlimit(10)

print 'Modified limit:', sys.getrecursionlimit()

def generate_recursion_error(i):
    print 'generate_recursion_error(%s)' % i
    generate_recursion_error(i+1)

try:
    generate_recursion_error(1)
except RuntimeError, err:
    print 'Caught exception:', err
```

一旦达到递归限制，解释器会产生一个 `RuntimeError` 异常，使程序有机会处理这种情况。

```
$ python sys_recursionlimit.py
```

```
Initial limit: 1000
Modified limit: 10
generate_recursion_error(1)
generate_recursion_error(2)
generate_recursion_error(3)
generate_recursion_error(4)
generate_recursion_error(5)
generate_recursion_error(6)
generate_recursion_error(7)
generate_recursion_error(8)
Caught exception: maximum recursion depth exceeded while getting
the str of an object
```

最大值

除了运行时可配置的值，`sys` 还包括一些变量，用来定义随系统不同而变化的一些类型的最大值。

```
import sys
```

```
print 'maxint      :', sys.maxint
print 'maxsize     :', sys.maxsize
print 'maxunicode:' , sys.maxunicode
```

maxint 是最大的可表示的常规整数。maxsize 是列表、字典、串或 C 解释器中 size 类型指示的其他数据结构的最大大小。maxunicode 是当前配置的解释器支持的最大整数 Unicode 值。

```
$ python sys_maximums.py
```

```
maxint      : 9223372036854775807
maxsize     : 9223372036854775807
maxunicode: 65535
```

浮点值

结构 float_info 包含解释器所用的浮点类型表示（基于底层系统的 float 实现）的有关信息。

```
import sys

print 'Smallest difference (epsilon):', sys.float_info.epsilon
print
print 'Digits (dig)                        :', sys.float_info.dig
print 'Mantissa digits (mant_dig):', sys.float_info.mant_dig
print
print 'Maximum (max):', sys.float_info.max
print 'Minimum (min):', sys.float_info.min
print
print 'Radix of exponents (radix):', sys.float_info.radix
print
print 'Maximum exponent for radix (max_exp):', sys.float_info.max_exp
print 'Minimum exponent for radix (min_exp):', sys.float_info.min_exp
print
print 'Max. exponent power of 10 (max_10_exp):', \
      sys.float_info.max_10_exp
print 'Min. exponent power of 10 (min_10_exp):', \
      sys.float_info.min_10_exp
print
print 'Rounding for addition (rounds):', sys.float_info.rounds
```

这些值依赖于编译器和底层系统。下面这些例子在 OS X 10.6.5 上生成。

```
$ python sys_float_info.py
```

```
Smallest difference (epsilon): 2.22044604925e-16
```

```
Digits (dig)                : 15
Mantissa digits (mant_dig): 53
```

```
Maximum (max): 1.79769313486e+308
Minimum (min): 2.22507385851e-308
```

```
Radix of exponents (radix): 2
```

```
Maximum exponent for radix (max_exp): 1024
```

```

Minimum exponent for radix (min_exp): -1021

Max. exponent power of 10 (max_10_exp): 308
Min. exponent power of 10 (min_10_exp): -307

Rounding for addition (rounds): 1

```

参见:

本地编译器的 float.h C 头文件包含这些设置的更多详细信息。

字节序

byteorder 设置为内置字节序。

```
import sys
```

```
print sys.byteorder
```

这个 w 值可以是 big 表示大端 (big endian), 或者是 little 表示小端 (little endian)。

```
$ python sys_byteorder.py
```

```
little
```

参见:

Endianness (http://en.wikipedia.org/wiki/Byte_order) 大端和小端内存系统的描述。

array (2.2 节) 和 struct (2.6 节) 依赖于数据字节序的其他模块。

float.h 本地编译器的 C 头文件包含有关这些设置的更多详细信息。

17.2.4 异常处理

sys 包含一些特性来捕获和处理异常。

未处理异常

很多应用的结构都包括一个主循环, 将执行包围在一个全局异常处理程序中, 来捕获较低层次未处理的错误。要达到同样的目的, 另一种方法是将 sys.excepthook 设置为一个函数, 它有 3 个参数 (错误类型、错误值和 traceback), 由这个函数来处理未处理的错误。

```
import sys
```

```
def my_excepthook(type, value, traceback):
    print 'Unhandled error:', type, value
```

```
sys.excepthook = my_excepthook
```

```
print 'Before exception'
```

```
raise RuntimeError('This is the error message')
```

```
print 'After exception'
```

由于产生异常的代码行未包围在 try:except 块中, 所以不会运行后面的 print 语句, 尽管这里设置了异常 hook (except hook)。

```
$ python sys_excepthook.py
```

```
Before exception
```

```
Unhandled error: <type 'exceptions.RuntimeError'> This is the error
message
```

当前异常

有些情况下, 不论是出于代码简洁性考虑, 还是为了避免与试图安装其自己的 excepthook 的库发生冲突, 使用显式的异常处理程序更合适。在这些情况下, 可以创建一个通用的处理函数, 通过调用 exc_info() 来获取线程的当前异常, 因此不需要显式地为它传递异常对象。

exc_info() 的返回值是一个包含 3 个成员的元组, 其中包含异常类、异常实例和 traceback。使用 exc_info() 要优于原来的形式 (使用 exc_type、exc_value 和 exc_traceback), 因为它是线程安全的。

```
import sys
import threading
import time

def do_something_with_exception():
    exc_type, exc_value = sys.exc_info()[:2]
    print 'Handling %s exception with message "%s" in %s' % \
        (exc_type.__name__, exc_value, threading.current_thread().name)

def cause_exception(delay):
    time.sleep(delay)
    raise RuntimeError('This is the error message')

def thread_target(delay):
    try:
        cause_exception(delay)
    except:
        do_something_with_exception()

threads = [ threading.Thread(target=thread_target, args=(0.3,)),
            threading.Thread(target=thread_target, args=(0.1,)),
            ]

for t in threads:
    t.start()
for t in threads:
    t.join()
```

这个例子通过忽略 exc_info() 的部分返回值, 避免了在 traceback 对象和当前帧中一个局部

变量之间引入循环引用。如果需要 traceback (例如, 以便记入日志), 可以显式地删除局部变量 (使用 del) 来避免循环。

```
$ python sys_exc_info.py
```

```
Handling RuntimeError exception with message "This is the error
message" in Thread-2
Handling RuntimeError exception with message "This is the error
message" in Thread-1
```

之前的交互式异常

交互式解释器中只有一个交互线程。该线程中的未处理异常会保存到 sys 的 3 个变量 (last_type、last_value 和 last_traceback) 中, 从而能轻松地获取来完成调试。通过使用 pdb 中的事后剖析调试工具, 则不再需要直接使用这些值。

```
$ python
```

```
Python 2.7 (r27:82508, Jul 3 2010, 21:12:11)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def cause_exception():
...     raise RuntimeError('This is the error message')
...
>>> cause_exception()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cause_exception
RuntimeError: This is the error message
>>> import pdb
>>> pdb.pm()
> <stdin>(2) cause_exception()
(Pdb) where
  <stdin>(1) <module>()
> <stdin>(2) cause_exception()
(Pdb)
```

参见:

exceptions (18.5 节) 内置错误。

pdb (16.6 节) Python 调试工具。

traceback (16.4 节) 处理 traceback 的模块。

17.2.5 底层线程支持

sys 包括一些底层函数来控制 and 调试线程行为。

检查间隔

Python 2 使用了一个全局锁, 以防止单独的线程破坏解释器状态。字节码执行会以一个固



定的间隔暂停，解释器则检查是否需要执行某个信号处理器。在这个间隔检查期间，当前线程还会释放全局解释器锁（global interpreter lock, GIL），然后重新请求，使其他线程有机会先获得这个锁来得到执行权。

默认的检查间隔是100字节码，可以用`sys.getcheckinterval()`得到当前值。用`sys.setcheckinterval()`改变这个间隔可能会对应用的性能产生影响，这取决于所完成的操作。

```
import sys
import threading
from Queue import Queue
import time

def show_thread(q, extraByteCodes):
    for i in range(5):
        for j in range(extraByteCodes):
            pass
        q.put(threading.current_thread().name)
    return

def run_threads(prefix, interval, extraByteCodes):
    print '%s interval = %s with %s extra operations' % \
        (prefix, interval, extraByteCodes)
    sys.setcheckinterval(interval)
    q = Queue()
    threads = [ threading.Thread(target=show_thread,
                                name='%s T%s' % (prefix, i),
                                args=(q, extraByteCodes)
                                )
               for i in range(3)
               ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    while not q.empty():
        print q.get()
    print
    return

run_threads('Default', interval=10, extraByteCodes=1000)
run_threads('Custom', interval=10, extraByteCodes=0)
```

检查间隔小于线程中的字节码数时，解释器会把控制权交给另一个线程，让它运行一段时间。这从第一组输出情况可以看出，其中检查间隔设置为100（默认值），对于i循环中的每一步要完成额外的1000次循环迭代。

另一方面，如果检查间隔大于线程所执行的字节码数，而且这个线程出于另外某种原因不能交出控制权，线程会在间隔到来之前完成工作。这种情况可以由第二个例子的队列中的name

值顺序来说明。

```
$ python sys_checkinterval.py

Default interval = 10 with 1000 extra operations
Default T0
Default T0
Default T0
Default T1
Default T2
Default T2
Default T0
Default T1
Default T2
Default T0
Default T1
Default T2
Default T1
Default T2
Default T1

Custom interval = 10 with 0 extra operations
Custom T0
Custom T0
Custom T0
Custom T0
Custom T0
Custom T1
Custom T1
Custom T1
Custom T1
Custom T1
Custom T1
Custom T2
Custom T2
Custom T2
Custom T2
```

修改检查间隔并不像看上去那么有用。还有很多其他因素也会控制 Python 线程的上下文切换行为。例如，如果一个线程完成 I/O，它会释放 GIL，可能因此允许另一个线程接管执行。

```
import sys
import threading
from Queue import Queue
import time

def show_thread(q, extraByteCodes):
    for i in range(5):
        for j in range(extraByteCodes):
```

```

        pass
        #q.put(threading.current_thread().name)
        print threading.current_thread().name
    return

def run_threads(prefix, interval, extraByteCodes):
    print '%s interval = %s with %s extra operations' % \
        (prefix, interval, extraByteCodes)
    sys.setcheckinterval(interval)
    q = Queue()
    threads = [ threading.Thread(target=show_thread,
                                name='%s T%s' % (prefix, i),
                                args=(q, extraByteCodes)
                                )
                for i in range(3)
                ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    while not q.empty():
        print q.get()
    print
    return

run_threads('Default', interval=100, extraByteCodes=1000)
run_threads('Custom', interval=10, extraByteCodes=0)

```

这个例子由第一个例子修改得来，以展示线程可以直接打印到 `sys.stdout`，而不是追加到一个队列。输出更不可预测。

```
$ python sys_checkinterval_io.py
```

```

Default interval = 100 with 1000 extra operations
Default T0
Default T1
Default T1Default T2

Default T0Default T2

Default T2
Default T2
Default T1
Default T2
Default T1
Default T1
Default T1
Default T0
Default T0

```




```

Default T0

Custom interval = 10 with 0 extra operations
Custom T0
Custom T0
Custom T0
Custom T0
Custom T1
Custom T1
Custom T1
Custom T1
Custom T2
Custom T2
Custom T2
Custom T1Custom T2

Custom T2

```

参见：

dis (18.3 节) 用 dis 模块分解 Python 代码，这是统计字节码的一种方法。

调试

找出死锁可能是处理线程最困难的方面之一。sys._current_frames() 能有所帮助，它能准确地显示出线程在哪里停止。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import sys
5  import threading
6  import time
7
8  io_lock = threading.Lock()
9  blocker = threading.Lock()
10
11 def block(i):
12     t = threading.current_thread()
13     with io_lock:
14         print '%s with ident %s going to sleep' % (t.name, t.ident)
15     if i:
16         blocker.acquire() # acquired but never released
17         time.sleep(0.2)
18     with io_lock:
19         print t.name, 'finishing'
20     return
21

```

```

22 # Create and start several threads that "block"
23 threads = [ threading.Thread(target=block, args=(i,)) for i in range(3) ]
24 for t in threads:
25     t.setDaemon(True)
26     t.start()
27
28 # Map the threads from their identifier to the thread object
29 threads_by_ident = dict((t.ident, t) for t in threads)
30
31 # Show where each thread is "blocked"
32 time.sleep(0.01)
33 with io_lock:
34     for ident, frame in sys._current_frames().items():
35         t = threads_by_ident.get(ident)
36         if not t:
37             # Main thread
38             continue
39         print t.name, 'stopped in', frame.f_code.co_name,
40         print 'at line', frame.f_lineno, 'of', frame.f_code.co_filename

```

`sys._current_frames()` 返回的字典以线程标识符为键，而不是线程名。需要稍做一点工作将这些标识符映射为线程对象。

由于 Thread-1 没有休眠，在检查其状态之前它就已经完成。由于它不再是活动的，所以不会出现在输出中。Thread-2 请求锁阻塞器（blocker），然后睡眠很短的一段时间。与此同时，Thread-3 尝试请求锁阻塞器，不过无法得到，因为已经被 Thread-2 占用。

```
$ python sys_current_frames.py
```

```

Thread-1 with ident 4300619776 going to sleep
Thread-1 finishing
Thread-2 with ident 4301156352 going to sleep
Thread-3 with ident 4302835712 going to sleep
Thread-3 stopped in block at line 16 of sys_current_frames.py
Thread-2 stopped in block at line 17 of sys_current_frames.py

```

参见：

`threading` (10.3 节) `threading` 模块包括一些类来创建 Python 线程。

`Queue` (2.5 节) `Queue` 模块提供了一个 FIFO 数据结构的线程安全实现。

Python Threads and the Global Interpreter Lock(<http://jessenoller.com/2009/02/01/python-threads-and-the-globalinterpreter-lock/>) Jesse Noller 的一篇文章，摘自 2007 年 12 月刊的《Python Magazine》。

Inside the Python GIL (www.dabeaz.com/python/GIL.pdf) 由 David Beazley 提供的演示文稿，描述了线程实现和性能问题，包括检查间隔与 GIL 的关联。

17.2.6 模块和导入

大多数 Python 程序最后都会是一个组合，包括多个模块以及导入这些模块的一个主应用。无论是否使用标准库的特性，还是将定制代码组织到单独的文件中以便于维护，理解和管理程序的依赖关系都是开发的一个重要方面。sys 包含了应用可用模块的有关信息，这些模块可能作为内置模块，也可能是导入的模块。sys 还定义了一些 hook 为特殊情况覆盖标准导入行为。

导入的模块

sys.modules 是一个字典，将所导入模块的名字映射为包含具体代码的模块对象。

```
import sys
import textwrap

names = sorted(sys.modules.keys())
name_text = ', '.join(names)

print textwrap.fill(name_text, width=65)
```

随着新模块的导入，sys.modules 的内容会改变。

```
$ python sys_modules.py
```

```
UserDict, __builtin__, __main__, _abcoll, _codecs, _sre,
_warnings, abc, codecs, copy_reg, encodings,
encodings.__builtin__, encodings.aliases, encodings.codecs,
encodings.encodings, encodings.utf_8, errno, exceptions,
genericpath, linecache, os, os.path, posix, posixpath, re,
signal, site, sre_compile, sre_constants, sre_parse, stat,
string, strop, sys, textwrap, types, warnings, zipimport
```

内置模块

Python 解释器编译时可以内置一些 C 模块，因此这些 C 模块不需要作为单独的共享库发布。这些模块不会出现在 sys.modules 管理的导入模块列表中，因为从理论上讲它们并不是导入的模块。要查找这些可用的内置模块，惟一的方法就是通过 sys.builtin_module_names。

```
import sys
import textwrap

name_text = ', '.join(sorted(sys.builtin_module_names))

print textwrap.fill(name_text, width=65)
```

这个脚本的输出可能会有变化，特别是用一个定制构建版本的解释器运行时。这个输出使用一个解释器的副本创建，该解释器由 OS X 的标准 python.org 安装程序安装。

```
$ python sys_builtins.py
```

```
__builtin__, __main__, _ast, _codecs, _sre, _symtable, _warnings,
```

```
errno, exceptions, gc, imp, marshal, posix, pwd, signal, sys,
thread, xxsubtype, zipimport
```

参见:

Build Instructions (<http://svn.python.org/view/python/trunk/README?view=markup>) 构建 Python 的说明, 摘自随源代码发布的 README。

导入路径

模块的搜索路径作为一个 Python 列表保存在 `sys.path` 中。这个路径的默认内容包括启动应用所用脚本的目录和当前工作目录。

```
import sys
```

```
for d in sys.path:
    print d
```

搜索路径中的第一个目录是示例脚本本身的主目录。后面是一系列平台特定的路径, 其中可能安装有已编译的扩展模块 (用 C 编写)。最后列出全局 `site-packages` 目录。

```
$ python sys_path_show.py

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/sys
.../lib/python2.7
.../lib/python2.7/plat-darwin
.../lib/python2.7/lib-tk
.../lib/python2.7/plat-mac
.../lib/python2.7/plat-mac/lib-scriptpackages
.../lib/python2.7/site-packages
```

通过将 shell 变量 `PYTHONPATH` 设置为一个用冒号分隔的目录列表, 从而在启动解释器之前可以修改导入搜索路径列表。

```
$ PYTHONPATH=/my/private/site-packages:/my/shared/site-packages \
> python sys_path_show.py
```

```
/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/sys
/my/private/site-packages
/my/shared/site-packages
.../lib/python2.7
.../lib/python2.7/plat-darwin
.../lib/python2.7/lib-tk
.../lib/python2.7/plat-mac
.../lib/python2.7/plat-mac/lib-scriptpackages
.../lib/python2.7/site-packages
```

程序还可以直接向 `sys.path` 添加元素来修改路径。

```
import sys
import os
```

```
base_dir = os.path.dirname(__file__) or '.'
```



```

print 'Base directory:', base_dir

# Insert the package_dir_a directory at the front of the path.
package_dir_a = os.path.join(base_dir, 'package_dir_a')
sys.path.insert(0, package_dir_a)

# Import the example module
import example
print 'Imported example from:', example.__file__
print '\t', example.DATA

# Make package_dir_b the first directory in the search path
package_dir_b = os.path.join(base_dir, 'package_dir_b')
sys.path.insert(0, package_dir_b)

# Reload the module to get the other version
reload(example)
print 'Reloaded example from:', example.__file__
print '\t', example.DATA

```

重新加载一个已经导入的模块时，会重新导入这个文件，并使用相同的 `module` 对象来保存结果。如果在第一次导入和 `reload()` 调用之间改变了路径，这意味着第二次可能加载一个不同的模块。

```
$ python sys_path_modify.py
```

```

Base directory: .
Imported example from: ./package_dir_a/example.pyc
    This is example A
Reloaded example from: ./package_dir_b/example.pyc
    This is example B

```

定制导入工具

通过修改搜索路径，程序员可以控制如何找到标准 Python 模块。不过，如果一个程序需要导入其他地方的代码，而不是从文件系统上常规的 `.py` 或 `.pyc` 文件导入，又该怎么做呢？PEP 302 解决了这个问题，引入了导入 hook（import hook）的思想，它会捕获到试图在搜索路径上查找一个模块，并采用候选策略从其他位置加载代码或者对其应用预处理。

可以通过两个不同阶段实现定制导入工具。查找工具（finder）负责找到一个模块，并提供一个加载工具（loader）来管理具体的导入。可以向 `sys.path_hooks` 列表追加一个工厂来添加定制模块查找工具。导入时，会把路径的各个部分提供给一个查找工具，直到一个部分声称支持（不产生 `ImportError`）。查找工具再负责搜索数据存储（由对应命名模块的路径入口表示）。

```

import sys

class NoisyImportFinder(object):

```

```

PATH_TRIGGER = 'NoisyImportFinder_PATH_TRIGGER'

def __init__(self, path_entry):
    print 'Checking %s:' % path_entry,
    if path_entry != self.PATH_TRIGGER:
        print 'wrong finder'
        raise ImportError()
    else:
        print 'works'
    return

def find_module(self, fullname, path=None):
    print 'Looking for "%s"' % fullname
    return None

sys.path_hooks.append(NoisyImportFinder)

sys.path.insert(0, NoisyImportFinder.PATH_TRIGGER)

try:
    import target_module
except Exception, e:
    print 'Import failed:', e

```

这个例子展示了如何实例化和查询查找工具。如果提供的路径入口与查找工具的特殊触发值不匹配，显然不是文件系统上的一个真正的路径，实例化时 Noisy-ImportFinder 会产生一个 ImportError。这个测试可以避免 NoisyImportFinder 破坏真正模块的导入。

```
$ python sys_path_hooks_noisy.py
```

```

Checking NoisyImportFinder_PATH_TRIGGER: works
Looking for "target_module"
Checking /Users/dhellmann/Documents/PyMOTW/book/PyMOTW/sys:
wrong finder
Import failed: No module named target_module

```

从 shelve 导入

查找工具找到一个模块时，它要负责返回一个能够导入该模块的加载工具（loader）。下面这个例子展示了一个定制导入工具，它会将其模块内容保存到由 shelve 创建的一个数据库中。

首先，使用脚本来用一个包（其中包含一个子模块和子包）填充这个 shelve。

```

import sys
import shelve
import os

filename = '/tmp/pymotw_import_example.shelve'

```

```

if os.path.exists(filename):
    os.unlink(filename)
db = shelve.open(filename)
try:
    db['data:README'] = """
=====
package README
=====

This is the README for ``package``.
"""
    db['package.__init__'] = """
print 'package imported'
message = 'This message is in package.__init__'
"""
    db['package.module1'] = """
print 'package.module1 imported'
message = 'This message is in package.module1'
"""
    db['package.subpackage.__init__'] = """
print 'package.subpackage imported'
message = 'This message is in package.subpackage.__init__'
"""
    db['package.subpackage.module2'] = """
print 'package.subpackage.module2 imported'
message = 'This message is in package.subpackage.module2'
"""
    db['package.with_error'] = """
print 'package.with_error being imported'
raise ValueError('raising exception to break import')
"""
    print 'Created %s with:' % filename
    for key in sorted(db.keys()):
        print '\t', key
finally:
    db.close()

```

一个真正的打包脚本会从文件系统读取内容，不过对于像这样一个简单的例子来说，使用硬编码的值就足够了。

```
$ python sys_shelve_importer_create.py
```

```

Created /tmp/pymotw_import_example.shelve with:
data:README
package.__init__
package.module1
package.subpackage.__init__
package.subpackage.module2
package.with_error

```

这个定制导入工具需要提供查找工具和加载工具，它们要知道如何在 shelf 中查找模块或包的源代码。

```
import contextlib
import imp
import os
import shelve
import sys

def _mk_init_name(fullname):
    """Return the name of the __init__ module
    for a given package name.
    """
    if fullname.endswith('.__init__'):
        return fullname
    return fullname + '.__init__'

def _get_key_name(fullname, db):
    """Look in an open shelve for fullname or
    fullname.__init__, return the name found.
    """
    if fullname in db:
        return fullname
    init_name = _mk_init_name(fullname)
    if init_name in db:
        return init_name
    return None

class ShelfFinder(object):
    """Find modules collected in a shelve archive."""

    def __init__(self, path_entry):
        if not os.path.isfile(path_entry):
            raise ImportError
        try:
            # Test the path_entry to see if it is a valid shelve
            with contextlib.closing(shelve.open(path_entry, 'r')):
                pass
        except Exception, e:
            raise ImportError(str(e))
        else:
            print 'shelve added to import path:', path_entry
            self.path_entry = path_entry
        return

    def __str__(self):
        return '<%s for "%s">' % (self.__class__.__name__,
                                   self.path_entry)
```



```

def find_module(self, fullname, path=None):
    path = path or self.path_entry
    print '\nlooking for "%s"\n in %s' % (fullname, path)
    with contextlib.closing(shelve.open(self.path_entry, 'r')
        ) as db:
        key_name = _get_key_name(fullname, db)
        if key_name:
            print ' found it as %s' % key_name
            return ShelveLoader(path)
    print ' not found'
    return None

class ShelveLoader(object):
    """Load source for modules from shelve databases."""
    def __init__(self, path_entry):
        self.path_entry = path_entry
        return

    def _get_filename(self, fullname):
        # Make up a fake filename that starts with the path entry
        # so pkgutil.get_data() works correctly.
        return os.path.join(self.path_entry, fullname)

    def get_source(self, fullname):
        print 'loading source for "%s" from shelf' % fullname
        try:
            with contextlib.closing(shelve.open(self.path_entry, 'r')
                ) as db:
                key_name = _get_key_name(fullname, db)
                if key_name:
                    return db[key_name]
            raise ImportError('could not find source for %s' %
                               fullname)
        except Exception, e:
            print 'could not load source:', e
            raise ImportError(str(e))

    def get_code(self, fullname):
        source = self.get_source(fullname)
        print 'compiling code for "%s"' % fullname
        return compile(source, self._get_filename(fullname),
                       'exec', dont_inherit=True)

    def get_data(self, path):
        print 'looking for data\n in %s\n for "%s"' % \
            (self.path_entry, path)
        if not path.startswith(self.path_entry):

```

```

        raise IOError
    path = path[len(self.path_entry)+1:]
    key_name = 'data:' + path
    try:
        with contextlib.closing(shelve.open(self.path_entry, 'r')
                                ) as db:
            return db[key_name]
    except Exception, e:
        # Convert all errors to IOError
        raise IOError
def is_package(self, fullname):
    init_name = _mk_init_name(fullname)
    with contextlib.closing(shelve.open(self.path_entry, 'r')
                            ) as db:
        return init_name in db

def load_module(self, fullname):
    source = self.get_source(fullname)

    if fullname in sys.modules:
        print 'reusing existing module from import of "%s"' % \
            fullname
        mod = sys.modules[fullname]
    else:
        print 'creating a new module object for "%s"' % fullname
        mod = sys.modules.setdefault(fullname,
                                     imp.new_module(fullname))

    # Set a few properties required by PEP 302
    mod.__file__ = self._get_filename(fullname)
    mod.__name__ = fullname
    mod.__path__ = self.path_entry
    mod.__loader__ = self
    mod.__package__ = '.'.join(fullname.split('.')[:-1])

    if self.is_package(fullname):
        print 'adding path for package'
        # Set __path__ for packages
        # so we can find the submodules.
        mod.__path__ = [ self.path_entry ]
    else:
        print 'imported as regular module'

    print 'execing source...'
    exec source in mod.__dict__
    print 'done'
    return mod

```

现在可以用 ShelfFinder 和 ShelfLoader 从一个 shelf 导入代码。这个例子显示了如何导入刚创建的 package。

```
import sys
import sys_shelve_importer
def show_module_details(module):
    print ' message      :', module.message
    print ' __name__     :', module.__name__
    print ' __package__ :', module.__package__
    print ' __file__      :', module.__file__
    print ' __path__      :', module.__path__
    print ' __loader__   :', module.__loader__

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelfFinder)
sys.path.insert(0, filename)

print 'Import of "package":'
import package

print
print 'Examine package details:'
show_module_details(package)

print
print 'Global settings:'
print 'sys.modules entry:'
print sys.modules['package']
```

修改路径之后，第一次出现导入时会把这个 shelf 添加到导入路径。查找工具找到这个 shelf，并返回一个加载工具，这个加载工具将用来完成从该 shelf 的所有导入。初始的包级导入会创建一个新的模块对象，然后使用 exec 运行从 shelf 加载的源代码。它将这个新模块用作命名空间，所以源代码中定义的名字可以作为模块级属性保留。

```
$ python sys_shelve_importer_package.py
```

```
Import of "package":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for "package"
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for "package" from shelf
creating a new module object for "package"
adding path for package
execing source...
package imported
done
```

Examine package details:

```
message      : This message is in package.__init__
__name__     : package
__package__  :
__file__     : /tmp/pymotw_import_example.shelve/package
__path__     : ['/tmp/pymotw_import_example.shelve']
__loader__   : <sys_shelve_importer.ShelveLoader object at 0x1006d42d0>
```

Global settings:

sys.modules entry:

```
<module 'package' from '/tmp/pymotw_import_example.shelve/package'>
```

定制包导入

可以采用同样的方式加载其他模块和子包。

```
import sys
import sys_shelve_importer

def show_module_details(module):
    print ' message      :', module.message
    print ' __name__     :', module.__name__
    print ' __package__  :', module.__package__
    print ' __file__     :', module.__file__
    print ' __path__     :', module.__path__
    print ' __loader__   :', module.__loader__

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print 'Import of "package.module1":'
import package.module1

print
print 'Examine package.module1 details:'
show_module_details(package.module1)

print
print 'Import of "package.subpackage.module2":'
import package.subpackage.module2

print
print 'Examine package.subpackage.module2 details:'
show_module_details(package.subpackage.module2)
```

查找工具接收要加载的模块的完整点名，并返回一个 ShelveLoader，它配置为从指向这个 shelf 文件的路径入口加载模块。模块的完全限定名传递给加载工具的 load_module() 方法，

它会构造并返回一个模块实例。

```
$ python sys_shelve_importer_module.py

Import of "package.module1":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for "package"
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for "package" from shelf
creating a new module object for "package"
adding path for package
execing source...
package imported
done

looking for "package.module1"
  in /tmp/pymotw_import_example.shelve
  found it as package.module1
loading source for "package.module1" from shelf
creating a new module object for "package.module1"
imported as regular module
execing source...
package.module1 imported
done

Examine package.module1 details:
  message      : This message is in package.module1
  __name__     : package.module1
  __package__  : package
  __file__     : /tmp/pymotw_import_example.shelve/package.module1
  __path__     : /tmp/pymotw_import_example.shelve
  __loader__   : <sys_shelve_importer.ShelveLoader object at 0x1006d42d0>
>

Import of "package.subpackage.module2":

looking for "package.subpackage"
  in /tmp/pymotw_import_example.shelve
  found it as package.subpackage.__init__
loading source for "package.subpackage" from shelf
creating a new module object for "package.subpackage"
adding path for package
execing source...
package.subpackage imported
```

```

done

looking for "package.subpackage.module2"
  in /tmp/pymotw_import_example.shelve
  found it as package.subpackage.module2
loading source for "package.subpackage.module2" from shelf
creating a new module object for "package.subpackage.module2"
imported as regular module
execing source...
package.subpackage.module2 imported
done

Examine package.subpackage.module2 details:
  message      : This message is in package.subpackage.module2
  __name__     : package.subpackage.module2
  __package__  : package.subpackage
  __file__     : /tmp/pymotw_import_example.shelve/package.subpackage.mo
dule2
  __path__     : /tmp/pymotw_import_example.shelve
  __loader__   : <sys_shelve_importer.ShelveLoader object at 0x1006d4390
>

```

在定制导入工具中重新加载模块

重新加载一个模块的处理稍有不同。不是创建一个新的模块对象，而会重用现有的模块。

```

import sys
import sys_shelve_importer
filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print 'First import of "package":'
import package

print
print 'Reloading "package":'
reload(package)

```

通过重用相同的对象，会保留这个模块现有的引用，即使类或函数定义已经因重新加载而修改。

```
$ python sys_shelve_importer_reload.py
```

```

First import of "package":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for "package"
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for "package" from shelf
creating a new module object for "package"

```

```

adding path for package
execing source...
package imported
done

```

Reloading "package":

```

looking for "package"
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for "package" from shelf
reusing existing module from import of "package"
adding path for package
execing source...
package imported
done

```

处理导入错误

任何查找工具都无法找到一个模块时，主导入代码会产生一个 `ImportError`。

```

import sys
import sys_shelve_importer

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

try:
    import package.module3
except ImportError, e:
    print 'Failed to import:', e

```

导入期间的其他错误会传播。

```
$ python sys_shelve_importer_missing.py
```

```

shelf added to import path: /tmp/pymotw_import_example.shelve

looking for "package"
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for "package" from shelf
creating a new module object for "package"
adding path for package
execing source...
package imported
done

looking for "package.module3"

```

```

in /tmp/pymotw_import_example.shelve
not found
Failed to import: No module named module3

```

包数据

除了定义 API 来加载可执行的 Python 代码外，PEP 302 还定义了一个可选的 API 来获取包数据，用于发布数据文件、文档和包使用的其他非代码资源。通过实现 `get_data()`，加载工具允许调用应用获取与包关联的数据，而不用考虑这个包具体如何安装（特别是不用假设这个包作为文件存储在一个文件系统上）。

```

import sys
import sys_shelve_importer
import os
import pkgutil

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

import package

readme_path = os.path.join(package.__path__[0], 'README')

readme = pkgutil.get_data('package', 'README')
# Equivalent to:
# readme = package.__loader__.get_data(readme_path)
print readme

foo_path = os.path.join(package.__path__[0], 'foo')
try:
    foo = pkgutil.get_data('package', 'foo')
    # Equivalent to:
    # foo = package.__loader__.get_data(foo_path)
except IOError as err:
    print 'ERROR: Could not load "foo"', err
else:
    print foo

```

`get_data()` 根据拥有数据的模块或包取一个路径。它将资源“文件”的内容作为一个字符串返回，或者如果这个资源不存在，则产生一个 `IOError`。

```
$ python sys_shelve_importer_get_data.py
```

```

shelf added to import path: /tmp/pymotw_import_example.shelve

looking for "package"
in /tmp/pymotw_import_example.shelve
found it as package.__init__

```



```

loading source for "package" from shelf
creating a new module object for "package"
adding path for package
execing source...
package imported
done
looking for data
  in /tmp/pymotw_import_example.shelve
  for "/tmp/pymotw_import_example.shelve/README"

```

```

=====
package README
=====

```

```
This is the README for ``package``.
```

```

looking for data
  in /tmp/pymotw_import_example.shelve
  for "/tmp/pymotw_import_example.shelve/foo"
ERROR: Could not load "foo"

```

参见:

pkgutil (19.3 节) 包含 `get_data()`，用于从一个包获取数据。

导入工具缓存

每次导入一个模块时都要搜索所有 hook，这样开销可能会很大。为了节省时间，会维护一个 `sys.path_importer_cache`，作为路径入口与加载工具（可以使用值来查找模块）之间的一个映射。

```

import sys

print 'PATH:'
for name in sys.path:
    if name.startswith(sys.prefix):
        name = '...' + name[len(sys.prefix):]
    print ' ', name

print
print 'IMPORTERS:'
for name, cache_value in sys.path_importer_cache.items():
    name = name.replace(sys.prefix, '...')
    print ' %s: %r' % (name, cache_value)

```

缓存值 `None` 表示使用默认的文件系统加载工具。路径上不存在的目录与一个 `imp.NullImporter` 实例关联，因为它们不能用来导入模块。在示例输出中，使用了多个 `zipimport.zipimporter` 实例来管理路径上找到的 EGG 文件。

```
$ python sys_path_importer_cache.py
```

```
PATH:
```

```

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/sys
.../lib/python2.7/site-packages/distribute-0.6.10-py2.7.egg
.../lib/python2.7/site-packages/pip-0.7.2-py2.7.egg
.../lib/python2.7.zip
.../lib/python2.7
.../lib/python2.7/plat-darwin
.../lib/python2.7/plat-mac
.../lib/python2.7/plat-mac/lib-scriptpackages
.../lib/python2.7/lib-tk
.../lib/python2.7/lib-old
.../lib/python2.7/lib-dynload
.../lib/python2.7/site-packages

```

IMPORTERS:

```

sys_path_importer_cache.py: <imp.NullImporter object at 0x100d02080>
.../lib/python2.7.zip: <imp.NullImporter object at 0x100d02030>
.../lib/python2.7/lib-dynload: None
.../lib/python2.7/encodings: None
.../lib/python2.7: None
.../lib/python2.7/lib-old: None
.../lib/python2.7/site-packages: None
.../lib/python2.7/plat-darwin: None
.../lib/python2.7/: None
.../lib/python2.7/plat-mac/lib-scriptpackages: None
.../lib/python2.7/plat-mac: None
.../lib/python2.7/site-packages/pip-0.7.2-py2.7.egg: None
.../lib/python2.7/lib-tk: None
.../lib/python2.7/site-packages/distribute-0.6.10-py2.7.egg: None

```

元路径

`sys.meta_path` 进一步扩展了可能的导入来源，允许在扫描常规的 `sys.path` 之前先搜索查找工具。元路径上查找工具的 API 与常规路径上查找工具的 API 是一样的。区别在于元查找工具不限于 `sys.path` 中的一项，它可以搜索任何地方。

```

import sys
import sys_shelve_importer
import imp
class NoisyMetaImporterFinder(object):

    def __init__(self, prefix):
        print 'Creating NoisyMetaImporterFinder for %s' % prefix
        self.prefix = prefix
        return

    def find_module(self, fullname, path=None):
        print 'looking for "%s" with path "%s"' % (fullname, path)
        name_parts = fullname.split('.')

```

```
    if name_parts and name_parts[0] == self.prefix:
        print ' ... found prefix, returning loader'
        return NoisyMetaImporterLoader(path)
    else:
        print ' ... not the right prefix, cannot load'
        return None

class NoisyMetaImporterLoader(object):

    def __init__(self, path_entry):
        self.path_entry = path_entry
        return

    def load_module(self, fullname):
        print 'loading %s' % fullname
        if fullname in sys.modules:
            mod = sys.modules[fullname]
        else:
            mod = sys.modules.setdefault(fullname,
                                          imp.new_module(fullname))

        # Set a few properties required by PEP 302
        mod.__file__ = fullname
        mod.__name__ = fullname
        # always looks like a package
        mod.__path__ = [ 'path-entry-goes-here' ]
        mod.__loader__ = self
        mod.__package__ = '.'.join(fullname.split('.')[::-1])

        return mod

# Install the meta-path finder
sys.meta_path.append(NoisyMetaImporterFinder('foo'))

# Import some modules that are "found" by the meta-path finder
print
import foo

print
import foo.bar

# Import a module that is not found
print
try:
    import bar
except ImportError, e:
    pass
```

搜索 `sys.path` 之前，会询问元路径上的各个查找工具，所以总有机会让一个中心导入工具加载模块，而不必显式地修改 `sys.path`。一旦“找到”模块，加载工具 API 就会像常规加载工具一样正常工作（不过为简单起见，这个例子有所缩减）。

```
$ python sys_meta_path.py

Creating NoisyMetaImporter for foo

looking for "foo" with path "None"
... found prefix, returning loader
loading foo

looking for "foo.bar" with path "['path-entry-goes-here']"
... found prefix, returning loader
loading foo.bar

looking for "bar" with path "None"
... not the right prefix, cannot load
```

参见：

`imp` (19.1 节) `imp` 模块提供了导入工具使用的工具。

`Importlib` 创建定制导入工具的基类和其他工具。

The Quick Guide to Python Eggs (<http://peak.telecommunity.com/DevCenter/PythonEggs>) 处理 EGG 的 PEAK 文档。

Python 3 stdlib module “`importlib`” (<http://docs.python.org/py3k/library/importlib.html>)

Python 3.x 包括一些抽象基类，可以更容易地创建定制导入工具。

PEP 302 (www.python.org/dev/peps/pep-0302) 导入 hook。

`zipimport` 从 ZIP 归档实现导入 Python 模块。

Import this, that, and the other thing: custom importers (<http://us.pycon.org/2010/conference/talks/?filter=core>) Brett Cannon 提供的 Py-Con 2010 演示文稿。

17.2.7 跟踪程序运行情况

有两种方法注入代码，来监视一个程序的运行，包括跟踪（tracing）和性能分析（profiling）。它们很类似，不过分别有不同的用途，所以也有不同的约束。监视一个程序最容易也最低效的方法是通过一个跟踪 hook（trace hook），可以用它来编写一个调试工具、监视代码覆盖，或者达到其他目的。

可以向 `sys.settrace()` 传递一个回调函数修改跟踪 hook。这个回调接收 3 个参数：所运行代码的栈帧、命名通知类型的串，以及一个事件特定的参数值。表 17.2 列出了程序执行时出现的不同层次信息所对应的 7 个不同事件类型。

跟踪函数调用

每个函数调用之前会生成一个 call 事件。传入回调的帧可以用来查找正在调用哪个函数，

以及从哪里调用。

表 17.2 settrace() 的事件 hook

事 件	何时出现	参 数 值
call	执行函数之前	None
line	执行一行代码之前	None
return	函数返回之前	返回的值
exception	出现一个异常之后	(异常, 值, traceback) 元组
c_call	调用一个 C 函数之前	C 函数对象
c_return	C 函数返回之后	None
c_exception	C 函数抛出一个错误之后	None

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import sys
5
6  def trace_calls(frame, event, arg):
7      if event != 'call':
8          return
9      co = frame.f_code
10     func_name = co.co_name
11     if func_name == 'write':
12         # Ignore write() calls from print statements
13         return
14     func_line_no = frame.f_lineno
15     func_filename = co.co_filename
16     caller = frame.f_back
17     caller_line_no = caller.f_lineno
18     caller_filename = caller.f_code.co_filename
19     print 'Call to %s\n on line %s of %s\n from line %s of %s\n' % \
20         (func_name, func_line_no, func_filename,
21          caller_line_no, caller_filename)
22     return
23
24  def b():
25      print 'in b()\n'
26
27  def a():
28      print 'in a()\n'
29      b()
30
31  sys.settrace(trace_calls)
32  a()

```

这个例子忽略了 write() 调用，因为 print 会使用 write() 写至 sys.stdout。

```
$ python sys_settrace_call.py

Call to a
  on line 27 of sys_settrace_call.py
  from line 32 of sys_settrace_call.py

in a()

Call to b
  on line 24 of sys_settrace_call.py
  from line 29 of sys_settrace_call.py

in b()
```

函数内部跟踪

跟踪 hook 可以返回一个新 hook，并在新作用域中使用（局部跟踪函数）。例如，可以控制跟踪只在某些模块或函数中逐行运行。

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import sys
5
6  def trace_lines(frame, event, arg):
7      if event != 'line':
8          return
9      co = frame.f_code
10     func_name = co.co_name
11     line_no = frame.f_lineno
12     filename = co.co_filename
13     print ' %s line %s' % (func_name, line_no)
14
15  def trace_calls(frame, event, arg):
16     if event != 'call':
17         return
18     co = frame.f_code
19     func_name = co.co_name
20     if func_name == 'write':
21         # Ignore write() calls from print statements
22         return
23     line_no = frame.f_lineno
24     filename = co.co_filename
25     print 'Call to %s on line %s of %s' % \
26         (func_name, line_no, filename)
27     if func_name in TRACE_INTRO:
28         # Trace into this function
29         return trace_lines
30     return
```



```

31
32 def c(input):
33     print 'input =', input
34     print 'Leaving c()'
35
36 def b(arg):
37     val = arg * 5
38     c(val)
39     print 'Leaving b()'
40
41 def a():
42     b(2)
43     print 'Leaving a()'
44
45 TRACE_INT0 = ['b']
46
47 sys.settrace(trace_calls)
48 a()

```

在这个例子中，全局函数列表保存在变量 `TRACE_INT0` 中，所以 `trace_calls()` 运行时，它能返回 `trace_lines()` 来启用 `b()` 内部的跟踪。

```
$ python sys_settrace_line.py
```

```

Call to a on line 41 of sys_settrace_line.py
Call to b on line 36 of sys_settrace_line.py
  b line 37
  b line 38
Call to c on line 32 of sys_settrace_line.py
input = 10
Leaving c()
  b line 39
Leaving b()
Leaving a()

```

监视栈

使用 `hook` 的另一种有用的方法是跟踪正在调用哪些函数，以及它们的返回值是什么。要监视返回值，可以监视 `return` 事件。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import sys
5
6  def trace_calls_and_returns(frame, event, arg):
7      co = frame.f_code
8      func_name = co.co_name
9      if func_name == 'write':

```

```

10         # Ignore write() calls from print statements
11         return
12     line_no = frame.f_lineno
13     filename = co.co_filename
14     if event == 'call':
15         print 'Call to %s on line %s of %s' % (func_name,
16                                             line_no,
17                                             filename)
18         return trace_calls_and_returns
19     elif event == 'return':
20         print '%s => %s' % (func_name, arg)
21     return
22
23 def b():
24     print 'in b()'
25     return 'response_from_b '
26
27 def a():
28     print 'in a()'
29     val = b()
30     return val * 2
31
32 sys.settrace(trace_calls_and_returns)
33 a()

```

局部跟踪函数用于监视返回事件，这说明调用一个函数时 `trace_calls_and_returns()` 需要返回其自身的一个引用，以便监视返回值。

```
$ python sys_settrace_return.py
```

```

Call to a on line 27 of sys_settrace_return.py
in a()
Call to b on line 23 of sys_settrace_return.py
in b()
b => response_from_b
a => response_from_b response_from_b

```

异常传播

可以通过在一个局部跟踪函数中查找 `exception` 事件来监视异常。出现异常时，会调用跟踪 hook 并提供一个元组，其中包含异常类型、异常对象和一个 `traceback` 对象。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import sys
5
6  def trace_exceptions(frame, event, arg):
7      if event != 'exception':

```



```

8         return
9         co = frame.f_code
10        func_name = co.co_name
11        line_no = frame.f_lineno
12        filename = co.co_filename
13        exc_type, exc_value, exc_traceback = arg
14        print 'Tracing exception:\n%s "%s"\non line %s of %s\n' % \
15              (exc_type.__name__, exc_value, line_no, func_name)
16
17    def trace_calls(frame, event, arg):
18        if event != 'call':
19            return
20        co = frame.f_code
21        func_name = co.co_name
22        if func_name in TRACE_INT0:
23            return trace_exceptions
24
25    def c():
26        raise RuntimeError('generating exception in c()')
27
28    def b():
29        c()
30        print 'Leaving b()'
31
32    def a():
33        b()
34        print 'Leaving a()'
35
36    TRACE_INT0 = ['a', 'b', 'c']
37
38    sys.settrace(trace_calls)
39    try:
40        a()
41    except Exception, e:
42        print 'Exception handler:', e

```

要注意应用局部函数的限制，因为格式化错误消息的一些内部函数会生成并忽略它们自己的异常。不论调用者是否捕获和忽略异常，每个异常都会被跟踪 hook 看到。

```
$ python sys_settrace_exception.py
```

```

Tracing exception:
RuntimeError "generating exception in c()"
on line 26 of c

```

```

Tracing exception:
RuntimeError "generating exception in c()"
on line 29 of b

```

```
Tracing exception:
RuntimeError "generating exception in c()"
on line 33 of a
```

```
Exception handler: generating exception in c()
```

参见:

profile (16.8 节) profile 模块文档介绍了如何使用一个现成的性能分析工具。

trace (16.7 节) trace 模块实现了多个代码分析特性。

Types and Members (<http://docs.python.org/library/inspect.html#typesand-members>) 帧和代码对象及其属性的描述。

Tracing python code (www.dalkescientific.com/writings/diary/archive/2005/04/20/tracing_python_code.html) 另一个 settrace() 教程。

Wicked hack: Python bytecode tracing (http://nedbatchelder.com/blog/200804/wicked_hack_python_bytecode_tracing.html) Ned Batchelder 完成的试验, 用比源代码行级更细的粒度进行跟踪。

17.3 os——可移植访问操作系统特定特性

作用: 可移植访问操作系统特定特性。

Python 版本: 1.4 及以后版本

os 模块为平台特定的模块 (如 posix、nt 和 mac) 提供了一个包装器。所有平台上函数的 API 都是相同的, 所以使用 os 模块可以提供一定的可移植性。不过, 并不是所有函数在每一个平台上都可用。这个总结中介绍的许多进程管理函数就对 Windows 不适用。

os 模块的 Python 文档的子标题是“杂类操作系统接口”。这个模块主要包括创建和管理运行进程或文件系统内容 (文件和目录) 的函数, 只有很少涉及其他功能。

17.3.1 进程所有者

os 提供的第一组函数用于确定和改变进程所有者 id。守护进程或一些特殊系统程序 (需要改变权限级别而不是作为 root 运行) 的作者最常使用这些函数。本节不打算解释 UNIX 安全、进程所有者等等的所有复杂细节。有关的更多详细内容请参考本节最后的参考列表。

下面的例子显示了一个进程的实际有效的用户和组信息, 然后改变这些有效值。这类似于系统自引导期间一个守护进程作为 root 启动时所要做的工作, 以降低权限等级, 作为一个不同的用户运行。

注意: 运行这个例子之前, 要改变 TEST_GID 和 TEST_UID 值, 使之对应一个真实用户。

```
import os
```

```
TEST_GID=501
TEST_UID=527

def show_user_info():
    print 'User (actual/effective) : %d / %d' % \
        (os.getuid(), os.geteuid())
    print 'Group (actual/effective) : %d / %d' % \
        (os.getgid(), os.getegid())
    print 'Actual Groups :', os.getgroups()
    return

print 'BEFORE CHANGE:'
show_user_info()
print

try:
    os.setegid(TEST_GID)
except OSError:
    print 'ERROR: Could not change effective group. Rerun as root.'
else:
    print 'CHANGED GROUP:'
    show_user_info()
    print

try:
    os.seteuid(TEST_UID)
except OSError:
    print 'ERROR: Could not change effective user. Rerun as root.'
else:
    print 'CHANGE USER:'
    show_user_info()
    print
```

在 OS X 上, 作为 id 为 527、组为 501 的用户运行时, 会生成以下输出。

```
$ python os_process_user_example.py
```

```
BEFORE CHANGE:
User (actual/effective) : 527 / 527
Group (actual/effective) : 501 / 501
Actual Groups : [501, 102, 204, 100, 98, 80, 61, 12, 500, 101]

CHANGED GROUP:
User (actual/effective) : 527 / 527
Group (actual/effective) : 501 / 501
Actual Groups : [501, 102, 204, 100, 98, 80, 61, 12, 500, 101]

CHANGE USER:
```

```
User (actual/effective) : 527 / 527
Group (actual/effective) : 501 / 501
Actual Groups : [501, 102, 204, 100, 98, 80, 61, 12, 500, 101]
```

这些值不会改变，因为只要不作为 root 运行，进程就不能改变其有效所有者值。试图将有效用户 id 或组 id 设置为非当前用户的其他值时，会导致一个 `OSError`。使用 `sudo` 运行同样的脚本，使它启动时有 root 权限，其结果则会完全不同。

```
$ sudo python os_process_user_example.py
```

```
BEFORE CHANGE:
```

```
User (actual/effective) : 0 / 0
Group (actual/effective) : 0 / 0
Actual Groups : [0, 204, 100, 98, 80, 61, 29, 20, 12, 9, 8,
5, 4, 3, 2, 1]
```

```
CHANGED GROUP:
```

```
User (actual/effective) : 0 / 0
Group (actual/effective) : 0 / 501
Actual Groups : [501, 204, 100, 98, 80, 61, 29, 20, 12, 9,
8, 5, 4, 3, 2, 1]
```

```
CHANGE USER:
```

```
User (actual/effective) : 0 / 527
Group (actual/effective) : 0 / 501
Actual Groups : [501, 204, 100, 98, 80, 61, 29, 20, 12, 9,
8, 5, 4, 3, 2, 1]
```

在这种情况下，由于它作为 root 启动，脚本可以改变进程的有效用户和组。一旦改变了有效 UID，进程则限于该用户的权限。由于非根用户不能改变其有效组，程序在改变用户之前需要先改变组。

17.3.2 进程环境

操作系统通过 `os` 模块为程序提供的另一个特性是环境。环境中设置的变量作为字符串可见，这些字符串可以通过 `os.environ` 或 `getenv()` 读取。环境变量通常用于配置值，如搜索路径、文件位置和调试标志。下面这个例子显示了如何获取一个环境变量，并将一个值传递到一个子进程。

```
import os

print 'Initial value:', os.environ.get('TESTVAR', None)
print 'Child process:'
os.system('echo $TESTVAR')

os.environ['TESTVAR'] = 'THIS VALUE WAS CHANGED'

print
print 'Changed value:', os.environ['TESTVAR']
```

```

print 'Child process:'
os.system('echo $TESTVAR')

del os.environ['TESTVAR']

print
print 'Removed value:', os.environ.get('TESTVAR', None)
print 'Child process:'
os.system('echo $TESTVAR')

```

os.environ 对象采用标准 Python 映射 API 来获取和设置值。对 os.environ 的改变会导出到子进程。

```
$ python -u os_environ_example.py
```

```
Initial value: None
```

```
Child process:
```

```
Changed value: THIS VALUE WAS CHANGED
```

```
Child process:
```

```
THIS VALUE WAS CHANGED
```

```
Removed value: None
```

```
Child process:
```

17.3.3 进程工作目录

如果操作系统有层次结构的文件系统，会有一个“当前工作目录”（current working directory）的概念：使用相对路径访问文件时，进程将使用文件系统上的这个目录作为起始位置。当前工作目录可以用 getcwd() 获取，用 chdir() 改变。

```

import os

print 'Starting:', os.getcwd()

print 'Moving up one:', os.pardir
os.chdir(os.pardir)

print 'After move:', os.getcwd()

```

利用 os.getcwd() 和 os.pardir 可以采用一种可移植的方式指示当前目录和父目录。

```
$ python os_cwd_example.py
```

```
Starting: /Users/dhellmann/Documents/PyMOTW/book/PyMOTW/os
```

```
Moving up one: ..
```

```
After move: /Users/dhellmann/Documents/PyMOTW/book/PyMOTW
```

17.3.4 管道

os 模块提供了很多函数用于使用管道管理子进程的 I/O。这些函数基本上都以相同的

方式工作，不过会根据所需的输入或输出类型返回不同的文件句柄。大多数情况下，由于 subprocess 模块（Python 2.4 中添加）的出现，这些函数已经变得过时，不过可能还有一些遗留代码在使用这些函数。

最常用的管道函数是 popen()。它创建一个新进程运行指定命令，并根据模式（mode）参数，为该进程的输入或输出关联一个流。

注意：尽管 popen() 函数在 Windows 上也可用，不过这里的例子都假设使用一个类 UNIX 的 shell。

```
import os

print 'popen, read:'
stdout = os.popen('echo "to stdout"', 'r')
try:
    stdout_value = stdout.read()
finally:
    stdout.close()
print '\tstdout:', repr(stdout_value)

print '\npopen, write:'
stdin = os.popen('cat -', 'w')
try:
    stdin.write('\tstdin: to stdin\n')
finally:
    stdin.close()
```

流的描述也采用类 UNIX 的术语。

- stdin——进程的“标准输入”流（文件描述符 0），进程可读，通常为终端输入。
- stdout——进程的“标准输出”流（文件描述符 1），进程可写，用于向用户显示常规输出。
- stderr——进程的“标准错误”流（文件描述符 2），进程可写，用于显示错误消息。

```
$ python -u os_popen.py
```

```
popen, read:
    stdout: 'to stdout\n'

popen, write:
    stdin: to stdin
```

调用者只能读写与子进程关联的流，这会限制其用途。子进程的其他文件描述符从父进程继承，所以第二个例子中 cat - 命令的输出出现在控制台上，因为其标准输出文件描述符与父脚本使用的文件描述符相同。

其他 popen() 变种可以提供另外的流，从而可以根据需要处理 stdin、stdout 和 stderr。例如，popen2() 返回一个只写的流，与子进程的 stdin 关联，另外返回一个只读的流，与其 stdout 关联。

```
import os
```

```

print 'popen2:'
stdin, stdout = os.popen2('cat -')
try:
    stdin.write('through stdin to stdout')
finally:
    stdin.close()
try:
    stdout_value = stdout.read()
finally:
    stdout.close()
print '\tpass through:', repr(stdout_value)

```

这个简化的例子展示了一个双向通信。写至 `stdin` 的值由 `cat` 读取（因为有 `-` 参数），然后写回到 `stdout`。更复杂的进程可以通过管道来回传递其他类型的消息——甚至可以是串行化的对象。

```
$ python -u os_popen2.py
```

```

popen2:
    pass through: 'through stdin to stdout'

```

大多数情况下，需要同时访问 `stdout` 和 `stderr`。`stdout` 流用于消息传递，`stderr` 流则用于传递错误。分别读取这两个流可以减少解析错误消息的复杂性。`popen3()` 函数返回 3 个打开的流，分别绑定到新进程的 `stdin`、`stdout` 和 `stderr`。

```

import os

print 'popen3:'
stdin, stdout, stderr = os.popen3('cat -; echo ";to stderr" 1>&2')
try:
    stdin.write('through stdin to stdout')
finally:
    stdin.close()
try:
    stdout_value = stdout.read()
finally:
    stdout.close()
print '\tpass through:', repr(stdout_value)
try:
    stderr_value = stderr.read()
finally:
    stderr.close()
print '\tstderr:', repr(stderr_value)

```

这个程序必须单独地读取和关闭 `stdout` 和 `stderr`。处理多进程的 I/O 时，会存在一些与流控制和顺序化有关的问题。I/O 会缓冲，如果调用者希望能够从一个流读取所有数据，子进程就必须关闭这个流来指示文件末尾。有关这些问题的更多信息，请参考 Python 库文档中“流控制问题”一节。

```
$ python -u os_popen3.py
```

```
popen3:
    pass through: 'through stdin to stdout'
    stderr: ';to stderr\n'
```

最后, `popen4()` 会返回两个流: `stdin` 和一个组合的 `stdout/stderr`。如果需要记录命令的结果, 但不用直接解析, 这个方法会很有用。

```
import os

print 'popen4:'
stdin, stdout_and_stderr = os.popen4('cat -; echo ";to stderr" 1>&2')
try:
    stdin.write('through stdin to stdout')
finally:
    stdin.close()
try:
    stdout_value = stdout_and_stderr.read()
finally:
    stdout_and_stderr.close()
print '\tcombined output:', repr(stdout_value)
```

写至 `stdout` 和 `stderr` 的所有消息都一起读取。

```
$ python -u os_popen4.py
```

```
popen4:
    combined output: 'through stdin to stdout;to stderr\n'
```

除了接受一个单字符串命令交给 `shell` 解析外, `popen2()`、`popen3()` 和 `popen4()` 还接受一个字符串序列, 其中包含命令及其参数。

```
import os

print 'popen2, cmd as sequence:'
stdin, stdout = os.popen2(['cat', '-'])
try:
    stdin.write('through stdin to stdout')
finally:
    stdin.close()
try:
    stdout_value = stdout.read()
finally:
    stdout.close()
print '\tpass through:', repr(stdout_value)
```

参数作为一个列表而不是单个字符串传递时, 运行命令之前 `shell` 不会处理这些参数。

```
$ python -u os_popen2_seq.py
```



```
popen2, cmd as sequence:
    pass through: 'through stdin to stdout'
```

17.3.5 文件描述符

os 包括一组标准函数来处理底层文件描述符 (file descriptor, 这是一个整数, 表示当前进程所拥有的打开文件)。这个 API 比 file 对象提供的 API 更底层。这里不再介绍这些函数, 因为直接使用 file 对象通常更容易。有关详细内容请参考库文档。

17.3.6 文件系统权限

关于文件的详细信息可以使用 stat() 或 lstat() 访问 (这个方法用于检查一个可能是符号链接的对象的状态)。

```
import os
import sys
import time
if len(sys.argv) == 1:
    filename = __file__
else:
    filename = sys.argv[1]

stat_info = os.stat(filename)

print 'os.stat(%s):' % filename
print '\tSize:', stat_info.st_size
print '\tPermissions:', oct(stat_info.st_mode)
print '\tOwner:', stat_info.st_uid
print '\tDevice:', stat_info.st_dev
print '\tLast modified:', time.ctime(stat_info.st_mtime)
```

取决于示例代码如何安装, 输出可能有变化。可以尝试在命令行上向 os_stat.py 传递不同的文件名。

```
$ python os_stat.py
```

```
os.stat(os_stat.py):
  Size: 1516
Permissions: 0100644
  Owner: 527
Device: 234881026
Last modified: Sun Nov 14 09:40:36 2010
```

在类 UNIX 的系统上, 可以使用 chmod() 改变文件权限, 只需传入模式 (一个整数)。模式值可以使用 stat 模块中定义的常量来构造。下面这个例子会来回切换用户的执行权限位。

```
import os
import stat

filename = 'os_stat_chmod_example.txt'
```

```

if os.path.exists(filename):
    os.unlink(filename)
with open(filename, 'wt') as f:
    f.write('contents')

# Determine what permissions are already set using stat
existing_permissions = stat.S_IMODE(os.stat(filename).st_mode)

if not os.access(filename, os.X_OK):
    print 'Adding execute permission'
    new_permissions = existing_permissions | stat.S_IXUSR
else:
    print 'Removing execute permission'
    # use xor to remove the user execute permission
    new_permissions = existing_permissions ^ stat.S_IXUSR

os.chmod(filename, new_permissions)

```

这个脚本假设有必要的权限可以在运行时修改文件的模式。

```
$ python os_stat_chmod.py
```

```
Adding execute permission
```

17.3.7 目录

很多函数可以用来处理文件系统上的目录，包括创建内容、列出内容和删除内容。

```

import os

dir_name = 'os_directories_example'

print 'Creating', dir_name
os.makedirs(dir_name)

file_name = os.path.join(dir_name, 'example.txt')
print 'Creating', file_name
with open(file_name, 'wt') as f:
    f.write('example file')

print 'Listing', dir_name
print os.listdir(dir_name)

print 'Cleaning up'
os.unlink(file_name)
os.rmdir(dir_name)

```

有两组函数用来创建和删除目录。创建一个目录可以使用 `makedirs()`，所有父目录都必须已经存在。用 `rmdir()` 删除一个目录时，实际上只会删除叶子目录（路径的最后部分）。与此相反，

`makedirs()` 和 `removedirs()` 要处理路径中的所有节点。`makedirs()` 会创建路径上所有不存在的部分, `removedirs()` 将删除所有父目录 (只要它们为空)。

```
$ python os_directories.py

Creating os_directories_example
Creating os_directories_example/example.txt
Listing os_directories_example
['example.txt']
Cleaning up
```

17.3.8 符号链接

对于支持符号链接的平台和文件系统, 还有一些函数来处理符号链接 (`symlink`)。

```
import os

link_name = '/tmp/' + os.path.basename(__file__)

print 'Creating link %s -> %s' % (link_name, __file__)
os.symlink(__file__, link_name)

stat_info = os.lstat(link_name)
print 'Permissions:', oct(stat_info.st_mode)

print 'Points to:', os.readlink(link_name)

# Cleanup
os.unlink(link_name)
```

使用 `symlink()` 可以创建一个符号链接, 使用 `readlink()` 可以读取符号链接来确定由该链接指示的原始文件。`lstat()` 函数类似于 `stat()`, 不过它处理的是符号链接。

```
$ python os_symlinks.py

Creating link /tmp/os_symlinks.py -> os_symlinks.py
Permissions: 0120755
Points to: os_symlinks.py
```

17.3.9 遍历目录树

函数 `walk()` 会递归地遍历一个目录, 对于每个目录, 会生成一个 `tuple`, 其中包含目录路径、该路径的所有直接子目录, 以及该目录中所有文件的文件名列表。

```
import os, sys

# If we are not given a path to list, use /tmp
if len(sys.argv) == 1:
    root = '/tmp'
else:
```

```
root = sys.argv[1]

for dir_name, sub_dirs, files in os.walk(root):
    print dir_name
    # Make the subdirectory names stand out with /
    sub_dirs = [ '%s/' % n for n in sub_dirs ]
    # Mix the directory contents together
    contents = sub_dirs + files
    contents.sort()
    # Show the contents
    for c in contents:
        print '\t%s' % c
    print
```

这个例子显示了一个递归的目录清单。

```
$ python os_walk.py ../zipimport
```

```
../zipimport
__init__.py
__init__.pyc
example_package/
index.rst
zipimport_example.zip
zipimport_find_module.py
zipimport_find_module.pyc
zipimport_get_code.py
zipimport_get_code.pyc
zipimport_get_data.py
zipimport_get_data.pyc
zipimport_get_data_nozip.py
zipimport_get_data_nozip.pyc
zipimport_get_data_zip.py
zipimport_get_data_zip.pyc
zipimport_get_source.py
zipimport_get_source.pyc
zipimport_is_package.py
zipimport_is_package.pyc
zipimport_load_module.py
zipimport_load_module.pyc
zipimport_make_example.py
zipimport_make_example.pyc

../zipimport/example_package
README.txt
__init__.py
__init__.pyc
```



17.3.10 运行外部命令

警告：处理进程的很多函数可移植性都很有限。要采用一种更一致的方法以一种平台独立的方式处理进程，请参考 subprocess 模块。

要运行一个单独的命令而不与之交互，最基本的方法就是使用 `system()`。它有一个单字符串参数，也就是命令行，将由运行一个 shell 的子进程执行。

```
import os

# Simple command
os.system('pwd')
```

`system()` 的返回值是运行这个程序的 shell 的退出值，包装为一个 16 位数字，高字节为退出状态，低字节是导致进程结束的信号数或 0。

```
$ python -u os_system_example.py

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/os
```

由于命令直接传递给 shell 来处理，它可以包含 shell 语法，如文件名模式匹配或环境变量。

```
import os

# Command with shell expansion
os.system('echo $TMPDIR')
```

shell 运行命令行时，这个串中的环境变量 `$TMPDIR` 会扩展。

```
$ python -u os_system_shell.py

/var/folders/9R/9Rlt+tR02Raxzk+F7lQ50U+++Uw/-Tmp-/
```

除非命令显式地在后台运行，否则 `system()` 调用会阻塞，直至完成。子进程的标准输入、输出和错误通道默认地绑定到调用者所拥有的相应流，不过可以使用 shell 语法重定向。

```
import os
import time

print 'Calling...'
os.system('date; (sleep 3; date) &')

print 'Sleeping...'
time.sleep(5)
```

不过这会让 shell 陷入麻烦，还有更好的方法来完成同样的工作。

```
$ python -u os_system_background.py

Calling...
Sat Dec  4 14:47:07 EST 2010
Sleeping...
Sat Dec  4 14:47:10 EST 2010
```



17.3.11 用 os.fork() 创建进程

POSIX 函数 `fork()` 和 `exec()` (Mac OS X、Linux 和其他 UNIX 系统上都可用) 通过 `os` 模块提供。有一些书专门介绍如何可靠地使用这些函数, 所以不要仅限于这里给出的介绍, 请参考库或者去书店了解更多细节。

要创建一个新进程, 作为当前进程的一个克隆, 可以使用 `fork()`。

```
import os

pid = os.fork()

if pid:
    print 'Child process id:', pid
else:
    print 'I am the child'
```

输出会根据每次运行例子时的系统状态而改变, 不过基本上类似于下面的输出。

```
$ python -u os_fork_example.py
```

```
I am the child
Child process id: 14133
```

创建之后, 这两个进程运行同样的代码。程序要想分辨在哪一个进程中, 需要检查 `fork()` 的返回值。如果值是 0, 当前进程是子进程; 如果不是 0, 说明程序在父进程中运行, 返回值就是子进程的进程 id。

父进程可以使用 `kill()` 和 `signal` 模块向子进程发送信号。首先, 定义一个信号处理程序, 收到这个信号时会调用这个处理程序。

```
import os
import signal
import time

def signal_usr1(signum, frame):
    "Callback invoked when a signal is received"
    pid = os.getpid()
    print 'Received USR1 in process %s' % pid
```

然后调用 `fork()`, 在父进程中, 使用 `kill()` 发送一个 USR1 信号之前会暂停很短一段时间。这个短暂的暂停使子进程有时间建立信号处理程序。

```
print 'Forking...'
child_pid = os.fork()
if child_pid:
    print 'PARENT: Pausing before sending signal...'
    time.sleep(1)
    print 'PARENT: Signaling %s' % child_pid
    os.kill(child_pid, signal.SIGUSR1)
```

在子进程中, 要建立信号处理程序, 并睡眠一段时间, 使父进程有时间发送信号。

```

else:
    print 'CHILD: Setting up signal handler'
    signal.signal(signal.SIGUSR1, signal_usr1)
    print 'CHILD: Pausing to wait for signal'
    time.sleep(5)

```

实际应用不需要（或不希望）调用 `sleep()`。

```
$ python os_kill_example.py
```

```

Forking...
PARENT: Pausing before sending signal...
PARENT: Signaling 14136
Forking...
CHILD: Setting up signal handler
CHILD: Pausing to wait for signal
Received USR1 in process 14136

```

要在子进程中处理不同行为，一种简单的方法是检查 `fork()` 的返回值并建立分支。要实现更复杂的行为，可能需要做更多代码分离而不只是一个简单的分支。另外一些情况下，还需要包装现有的程序。对于这两类情况，可以用 `exec*()` 系列的函数来运行另一个程序。

```

import os

child_pid = os.fork()
if child_pid:
    os.waitpid(child_pid, 0)
else:
    os.execvp('pwd', 'pwd', '-P')

```

由 `exec()` 运行程序时，该程序的代码会替换现有进程的代码。

```
$ python os_exec_example.py
```

```
/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/os
```

`exec()` 有很多变种，使用哪个变种取决于参数的形式、父进程的路径和环境是否要复制到子进程等等。对于所有这些变种，第一个参数都是路径或文件名，其余的参数将控制程序如何运行。它们可能作为命令行参数传递，或者会覆盖进程“环境”（参见 `os.environ` 和 `os.getenv`）。请参考库文档来全面了解有关的详细信息。

17.3.12 等待子进程

很多包含大量计算的程序会使用多个进程，来绕过 Python 的线程限制和全局解释器锁。启动多个进程运行不同任务时，主进程开始新的子进程之前需要等待一个或多个子进程完成，以避免服务器负载过大。对此有一些不同的方法，可以使用 `wait()` 和相关函数来实现。

如果哪个子进程最先退出并不重要，可以使用 `wait()`。一旦有子进程退出它就会返回。

```

import os
import sys

```

```
import time

for i in range(2):
    print 'PARENT %s: Forking %s' % (os.getpid(), i)
    worker_pid = os.fork()
    if not worker_pid:
        print 'WORKER %s: Starting' % i
        time.sleep(2 + i)
        print 'WORKER %s: Finishing' % i
        sys.exit(i)

for i in range(2):
    print 'PARENT: Waiting for %s' % i
    done = os.wait()
    print 'PARENT: Child done:', done
```

wait() 的返回值是一个元组，其中包含进程 id 和退出状态（组合为一个 16 位值）。退出状态的低字节是结束进程的信号编号，高字节是退出时进程返回的状态码。

```
$ python os_wait_example.py
```

```
PARENT 14154: Forking 0
PARENT 14154: Forking 1
WORKER 0: Starting
PARENT: Waiting for 0
WORKER 1: Starting
WORKER 0: Finishing
PARENT: Child done: (14155, 0)
PARENT: Waiting for 1
WORKER 1: Finishing
PARENT: Child done: (14156, 256)
```

要等待一个特定的进程，可以使用 waitpid()。

```
import os
import sys
import time

workers = []
for i in range(2):
    print 'PARENT %d: Forking %s' % (os.getpid(), i)
    worker_pid = os.fork()
    if not worker_pid:
        print 'WORKER %s: Starting' % i
        time.sleep(2 + i)
        print 'WORKER %s: Finishing' % i
        sys.exit(i)
    workers.append(worker_pid)

for pid in workers:
```




```

print 'PARENT: Waiting for %s' % pid
done = os.waitpid(pid, 0)
print 'PARENT: Child done:', done

```

传入目标进程的进程 id。waitpid() 会阻塞，直至该进程退出。

```
$ python os_waitpid_example.py
```

```

PARENT 14162: Forking 0
PARENT 14162: Forking 1
PARENT: Waiting for 14163
WORKER 0: Starting
WORKER 1: Starting
WORKER 0: Finishing
PARENT: Child done: (14163, 0)
PARENT: Waiting for 14164
WORKER 1: Finishing
PARENT: Child done: (14164, 256)

```

wait3() 和 wait4() 采用同样的方式，不过会返回关于子进程的更详细的信息，包括 pid、退出状态和资源使用情况。

17.3.13 Spawn

为提供便利，spawn() 系列函数可以在一个语句中完成 fork() 和 exec() 处理。

```

import os

os.spawnlp(os.P_WAIT, 'pwd', 'pwd', '-P')

```

第一个参数是一个模式，指示返回之前是否等待进程完成。这个例子会等待。使用 P_NOWAIT 允许另一个进程开始，不过然后会恢复执行当前进程。

```
$ python os_spawn_example.py
```

```
/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/os
```

17.3.14 文件系统权限

可以用函数 access() 检查一个进程对一个文件的访问权限。

```

import os

print 'Testing:', __file__
print 'Exists:', os.access(__file__, os.F_OK)
print 'Readable:', os.access(__file__, os.R_OK)
print 'Writable:', os.access(__file__, os.W_OK)
print 'Executable:', os.access(__file__, os.X_OK)

```

取决于示例代码如何安装，结果可能会有变化，不过输出可能与下面类似。

```
$ python os_access.py
```



```

Testing: os_access.py
Exists: True
Readable: True
Writable: True
Executable: False

```

`access()` 的库文档给出了两个特殊的警告。首先，对一个文件具体调用 `open()` 之前，调用 `access()` 来检查这个文件是否能够打开并没有太大意义。这两个调用之间有一个很小（但确实存在）的时间窗，在此期间文件权限可能改变。另一个警告主要适用于扩展了 POSIX 权限语义的网络文件系统。有些类型的文件系统响应 POSIX 调用时，可能会指出一个进程有权限访问一个文件，当试图通过 POSIX 调用出于某种未经测试的原因使用 `open()` 打开文件时，会报告一个失败。总之，最好基于所需的模式调用 `open()`，如果出现问题则捕获产生的 `IOError`。

参见：

`os` (<http://docs.python.org/lib/module-os.html>) 这个模块的标准库文档。

Flow Control Issues (<http://docs.python.org/library/popen2.html#popen2-flowcontrol>) `popen2()` 的标准库文档，并指出如何避免死锁。

`signal` (10.2 节) 关于 `signal` 模块的部分更详细地介绍信号处理技术。

`subprocess` (10.1 节) `subprocess` 模块取代了 `os.popen()`。

`multiprocessing` (10.4 节) `multiprocessing` 模块使额外进程的处理更为容易。

6.5.3 节 `shutil` 模块也包括一些处理目录树的函数。

`tempfile` (6.4 节) `tempfile` 模块用于处理临时文件。

UNIX Manual Page Introduction (www.scit.wlv.ac.uk/cgi-bin/mansec?2+intro) 包括实际和有效 id 等的定义。

Speaking UNIX, Part 8 (www.ibm.com/developerworks/aix/library/auspeakingunix8/index.html) 了解 UNIX 如何实现多任务。

UNIX Concepts (www.linuxhq.com/guides/LUG/node67.html) 关于 `stdin`、`stdout` 和 `stderr` 的更多讨论。

Delve into UNIX Process Creation (www.ibm.com/developerworks/aix/library/auunixprocess.html) 解释 UNIX 进程的生命周期。

《Advanced Programming in the UNIX(R) Environment》W. Richard Stevens 和 Stephen A. Rago 所著，由 Addison-Wesley Professional 于 2005 出版 (ISBN-10:0201433079)。介绍如何使用多进程，如处理信号、关闭重复的文件描述符等等。

17.4 platform——系统版本信息

作用：探查底层平台的硬件、操作系统和解释器版本信息。

Python 版本：2.3 及以后版本

尽管 Python 通常用作一个跨平台的语言，有时也有必要知道程序在何种系统上运行。构建工具需要这个信息，另外应用可能也需要知道它使用的一些库或外部命令在不同操作系统上有不同的

接口。例如，一个管理操作系统网络配置的工具可能对网络接口、别名和 IP 地址等等定义了可移植的表示。不过，编辑配置文件时，它必须对主机有更多了解，从而能使用正确的操作系统配置命令和文件。platform 模块包括有一些工具来了解运行程序的解释器、操作系统和硬件平台。

注意：本节的示例输出在 3 个系统上生成：一个是运行 OS X 10.6.5 的 MacBook Pro3.1、一个是运行 CentOS 5.5 的 VMware Fusion VM；还有一个是运行 Microsoft Windows 2008 的 Dell PC。Python 使用 python.org 的预编译安装工具安装在 OS X 和 Windows 系统上。Linux 系统在运行一个由源代码本地构建的解释器。

17.4.1 解释器

有 4 个函数可以得到当前 Python 解释器的有关信息。python_version() 和 python_version_tuple() 可以返回不同形式的解释器版本，包括主版本、次版本和补丁级组件。python_compiler() 会报告用来构建解释器的编译器。python_build() 将给出解释器构建的版本串。

```
import platform
```

```
print 'Version      :', platform.python_version()
print 'Version tuple:', platform.python_version_tuple()
print 'Compiler     :', platform.python_compiler()
print 'Build        :', platform.python_build()
```

OS X:

```
$ python platform_python.py
```

```
Version      : 2.7.0
Version tuple: ('2', '7', '0')
Compiler     : GCC 4.0.1 (Apple Inc. build 5493)
Build        : ('r27:82508', 'Jul  3 2010 21:12:11')
```

Linux:

```
$ python platform_python.py
```

```
Version      : 2.7.0
Version tuple: ('2', '7', '0')
Compiler     : GCC 4.1.2 20080704 (Red Hat 4.1.2-46)
Build        : ('r27', 'Aug 20 2010 11:37:51')
```

Windows:

```
C:> python.exe platform_python.py
```

```
Version      : 2.7.0
Version tuple: ['2', '7', '0']
Compiler     : MSC v.1500 64 bit (AMD64)
Build        : ('r27:82525', 'Jul  4 2010 07:43:08')
```



17.4.2 平台

`platform()` 函数返回一个字符串，其中包含一个通用的平台标识符。这个函数接受两个可选的布尔参数。如果 `aliased` 为 `True`，返回值中的名会从一个正式名转换为更常用的格式。如果 `terse` 为 `true`，则会返回一个最小值，即去除某些部分，而不是返回完整的串。

```
import platform
```

```
print 'Normal :', platform.platform()
print 'Aliased:', platform.platform(aliased=True)
print 'Terse  :', platform.platform(terse=True)
```

OS X:

```
$ python platform_platform.py
```

```
Normal : Darwin-10.5.0-i386-64bit
Aliased: Darwin-10.5.0-i386-64bit
Terse  : Darwin-10.5.0
```

Linux:

```
$ python platform_platform.py
```

```
Normal : Linux-2.6.18-194.3.1.el5-i686-with-redhat-5.5-Final
Aliased: Linux-2.6.18-194.3.1.el5-i686-with-redhat-5.5-Final
Terse  : Linux-2.6.18-194.3.1.el5-i686-with-glibc2.3
```

Windows:

```
C:> python.exe platform_platform.py
```

```
Normal : Windows-2008ServerR2-6.1.7600
Aliased: Windows-2008ServerR2-6.1.7600
Terse  : Windows-2008ServerR2
```

17.4.3 操作系统和硬件信息

还可以得到运行解释器的操作系统和硬件的更多详细信息。`uname()` 返回一个元组，其中包含系统、节点、发行号、版本、机器和处理器值。可以通过同名的函数访问各个值，如表 17.3 所示。

表 17.3 平台信息函数

函 数	返 回 值
<code>system()</code>	操作系统名
<code>node()</code>	服务器主机名，不是完全限定名
<code>release()</code>	操作系统发行号
<code>version()</code>	更详细的系统版本信息

(续)

函 数	返 回 值
machine()	硬件类型标识符, 如 'i386'
processor()	处理器实际标识符 (有些情况下与 machine() 值相同)

```
import platform
```

```
print 'uname:', platform.uname()
```

```
print
```

```
print 'system  :', platform.system()
```

```
print 'node    :', platform.node()
```

```
print 'release  :', platform.release()
```

```
print 'version  :', platform.version()
```

```
print 'machine  :', platform.machine()
```

```
print 'processor:', platform.processor()
```

OS X:

```
$ python platform_os_info.py
```

```
uname: ('Darwin', 'farnsworth.local', '10.5.0', 'Darwin Kernel
Version 10.5.0: Fri Nov  5 23:20:39 PDT 2010;
root:xnu-1504.9.17~1/RELEASE_I386', 'i386', 'i386')
```

```
system   : Darwin
node     : farnsworth.local
release  : 10.5.0
version  : Darwin Kernel Version 10.5.0: Fri Nov  5 23:20:39 PDT
2010; root:xnu-1504.9.17~1/RELEASE_I386
machine  : i386
processor: i386
```

Linux:

```
$ python platform_os_info.py
```

```
uname: ('Linux', 'hermes.hellfly.net', '2.6.18-194.3.1.el5',
'#1 SMP Thu May 13 13:09:10 EDT 2010', 'i686', 'i686')
```

```
system   : Linux
node     : hermes.hellfly.net
release  : 2.6.18-194.3.1.el5
version  : #1 SMP Thu May 13 13:09:10 EDT 2010
machine  : i686
processor: i686
```

Windows:

```
C:> python.exe platform_os_info.py
```

```

uname: ('Windows', 'dhellmann', '2008ServerR2', '6.1.7600',
'AMD64', 'Intel64 Family 6 Model 15 Stepping 11, GenuineIntel')

system      : Windows
node        : dhellmann
release     : 2008ServerR2
version     : 6.1.7600
machine     : AMD64
processor: Intel64 Family 6 Model 15 Stepping 11, GenuineIntel

```

17.4.4 可执行程序体系结构

可以使用 `architecture()` 函数查看程序的体系结构信息。第一个参数是可执行程序的路径（默认为 `sys.executable`，即 Python 解释器）。返回值是一个元组，包含位体系结构和使用的链接格式。

```

import platform

print 'interpreter:', platform.architecture()
print '/bin/ls      : ', platform.architecture('/bin/ls')

```

OS X:

```
$ python platform_architecture.py
```

```

interpreter: ('64bit', '')
/bin/ls      : ('64bit', '')

```

Linux:

```
$ python platform_architecture.py
```

```

interpreter: ('32bit', 'ELF')
/bin/ls      : ('32bit', 'ELF')

```

Windows:

```
C:> python.exe platform_architecture.py
```

```

interpreter : ('64bit', 'WindowsPE')
iexplore.exe : ('64bit', '')

```

参见:

`platform` (<http://docs.python.org/lib/module-platform.html>) 这个模块的标准库文档。

17.5 resource——系统资源管理

作用：管理 UNIX 程序的系统资源限制。

Python 版本：1.5.2 及以后版本

`resource` 中的函数可以检查一个进程消耗的当前系统资源，并做出限制，来控制一个程序给系统增加的负载。

17.5.1 当前使用情况

使用 `getrusage()` 查看当前进程和其子进程使用的资源。返回值是一个数据结构，其中包含当前系统状态下的一些资源度量。

注意：这里并没有显示所收集的全部资源值。更完整的列表可以参考 `resource` 的标准库文档。

```
import resource
import time

usage = resource.getrusage(resource.RUSAGE_SELF)
for name, desc in [
    ('ru_utime', 'User time'),
    ('ru_stime', 'System time'),
    ('ru_maxrss', 'Max. Resident Set Size'),
    ('ru_ixrss', 'Shared Memory Size'),
    ('ru_idrss', 'Unshared Memory Size'),
    ('ru_isrss', 'Stack Size'),
    ('ru_inblock', 'Block inputs'),
    ('ru_oublock', 'Block outputs'),
]:
    print '%-25s (%-10s) = %s' % (desc, name, getattr(usage, name))
```

由于这个测试程序极其简单，它没有使用太多资源。

```
$ python resource_getrusage.py
```

```
User time                (ru_utime ) = 0.013974
System time              (ru_stime ) = 0.013182
Max. Resident Set Size  (ru_maxrss ) = 5378048
Shared Memory Size      (ru_ixrss ) = 0
Unshared Memory Size    (ru_idrss ) = 0
Stack Size              (ru_isrss ) = 0
Block inputs            (ru_inblock) = 0
Block outputs           (ru_oublock) = 1
```

17.5.2 资源限制

除了当前实际使用的资源，还可以检查对应用施加的限制，然后加以修改。

```
import resource

print 'Resource limits (soft/hard):'
for name, desc in [
    ('RLIMIT_CORE', 'core file size'),
    ('RLIMIT_CPU', 'CPU time'),
```

```

('RLIMIT_FSIZE', 'file size'),
('RLIMIT_DATA', 'heap size'),
('RLIMIT_STACK', 'stack size'),
('RLIMIT_RSS', 'resident set size'),
('RLIMIT_NPROC', 'number of processes'),
('RLIMIT_NOFILE', 'number of open files'),
('RLIMIT_MEMLOCK', 'lockable memory address'),
]:
limit_num = getattr(resource, name)
soft, hard = resource.getrlimit(limit_num)
print '%-23s %s / %s' % (desc, soft, hard)

```

对应各个限制的返回值分别是一个元组，包含当前配置施加的软（soft）限制，以及由操作系统施加的硬（hard）限制。

```
$ python resource_getrlimit.py
```

```

Resource limits (soft/hard):
core file size          0 / 9223372036854775807
CPU time                9223372036854775807 / 9223372036854775807
file size               9223372036854775807 / 9223372036854775807
heap size               9223372036854775807 / 9223372036854775807
stack size              8388608 / 67104768
resident set size       9223372036854775807 / 9223372036854775807
number of processes     266 / 532
number of open files    7168 / 9223372036854775807
lockable memory address 9223372036854775807 / 9223372036854775807

```

可以用 `setrlimit()` 改变限制。

```

import resource
import os

soft, hard = resource.getrlimit(resource.RLIMIT_NOFILE)
print 'Soft limit starts as : ', soft

resource.setrlimit(resource.RLIMIT_NOFILE, (4, hard))

soft, hard = resource.getrlimit(resource.RLIMIT_NOFILE)
print 'Soft limit changed to : ', soft

random = open('/dev/random', 'r')
print 'random has fd =', random.fileno()
try:
    null = open('/dev/null', 'w')
except IOError, err:
    print err
else:
    print 'null has fd =', null.fileno()

```


这个例子使用 `RLIMIT_NOFILE` 来控制允许打开的文件数，将它改为比默认值小的一个软限制。

```
$ python resource_setrlimit_nofile.py

Soft limit starts as : 7168
Soft limit changed to : 4
random has fd = 3
[Errno 24] Too many open files: '/dev/null'
```

限制一个进程消耗的 CPU 时间可能也很有用，这样可以避免一个进程耗费太多 CPU 时间。当进程运行的时间超过了所分配的时间时，会向它发送一个 `SIGXCPU` 信号。

```
import resource
import sys
import signal
import time

# Set up a signal handler to notify us
# when we run out of time.
def time_expired(n, stack):
    print 'EXPIRED :', time.ctime()
    raise SystemExit('(time ran out)')

signal.signal(signal.SIGXCPU, time_expired)

# Adjust the CPU time limit
soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
print 'Soft limit starts as :', soft

resource.setrlimit(resource.RLIMIT_CPU, (1, hard))

soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
print 'Soft limit changed to :', soft
print

# Consume some CPU time in a pointless exercise
print 'Starting:', time.ctime()
for i in range(200000):
    for i in range(200000):
        v = i * i
# We should never make it this far
print 'Exiting :', time.ctime()
```

正常情况下，信号处理程序应当刷新输出所有打开的文件，然后将其关闭，不过在这里，它只是打印出一个消息然后退出。

```
$ python resource_setrlimit_cpu.py

Soft limit starts as : 9223372036854775807
```

```
Soft limit changed to : 1
```

```
Starting: Sat Dec 4 15:02:57 2010
EXPIRED : Sat Dec 4 15:02:58 2010
(time ran out)
```

参见:

resource (<http://docs.python.org/library/resource.html>) 这个模块的标准库文档。

signal (10.2 节) 提供了注册信号处理程序的详细内容。

17.6 gc——垃圾回收器

作用: 管理 Python 对象使用的内存。

Python 版本: 2.1 及以后版本

gc 提供了 Python 的底层内存管理机制, 即自动垃圾回收器。这个模块包括一些函数, 可以控制回收器如何操作, 以及如何检查系统已知的对象, 这些对象可能在等待收集或者陷入引用循环而无法释放。

17.6.1 跟踪引用

利用 gc, 可以使用对象之间来回的引用查找复杂数据结构中的循环。如果已知一个数据结构中存在循环, 可以使用定制代码来检查它的属性。如果循环在未知代码中, 则可以使用 `get_referents()` 和 `get_referrers()` 函数建立通用调试工具。

例如, `get_referents()` 显示了输入参数指示的对象。

```
import gc
import pprint

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):
        print 'Linking nodes %s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)
```

```

print
print 'three refers to:'
for r in gc.get_referents(three):
    pprint.pprint(r)

```

在这里，Graph 实例 three 包含其实例字典（__dict__ 属性中）以及其类的引用。

```
$ python gc_get_referents.py
```

```

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

```

```

three refers to:
{'name': 'three', 'next': Graph(one)}
<class '__main__.Graph'>

```

下一个例子使用一个 Queue 对所有对象引用完成一个广度优先遍历，查找循环。插入到队列中的元素为元组，其中包含目前为止的引用链和下一个要检查的对象。它从 three 开始，查看它引用的所有对象。跳过类可以避免查看方法、模块等等。

```

import gc
import pprint
import Queue

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):
        print 'Linking nodes %s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

print

seen = set()
to_process = Queue.Queue()

```

```

# Start with an empty object chain and Graph three.
to_process.put( ([], three) )

# Look for cycles, building the object chain for each object found
# in the queue so the full cycle can be printed at the end.
while not to_process.empty():
    chain, next = to_process.get()
    chain = chain[:]
    chain.append(next)
    print 'Examining:', repr(next)
    seen.add(id(next))
    for r in gc.get_referents(next):
        if isinstance(r, basestring) or isinstance(r, type):
            # Ignore strings and classes
            pass
        elif id(r) in seen:
            print
            print 'Found a cycle to %s:' % r
            for i, link in enumerate(chain):
                print '  %d: ' % i,
                pprint.pprint(link)
            else:
                to_process.put( (chain, r) )

```

通过监视已处理的对象，可以很容易地发现节点中的循环。为了避免保存这些对象的引用，它们的 `id()` 值会缓存在一个集合中。循环中找到的字典对象是 `Graph` 实例的 `__dict__` 值，其中包含它们的实例属性。

```
$ python gc_get_referents_cycles.py
```

```

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

Examining: Graph(three)
Examining: {'name': 'three', 'next': Graph(one)}
Examining: Graph(one)
Examining: {'name': 'one', 'next': Graph(two)}
Examining: Graph(two)
Examining: {'name': 'two', 'next': Graph(three)}

Found a cycle to Graph(three):
0: Graph(three)
1: {'name': 'three', 'next': Graph(one)}
2: Graph(one)
3: {'name': 'one', 'next': Graph(two)}
4: Graph(two)
5: {'name': 'two', 'next': Graph(three)}

```



17.6.2 强制垃圾回收

尽管解释器执行一个程序时会自动运行垃圾回收器，但是如果需要释放大量对象，或者如果当时没有太多工作，相应的回收器不会影响应用性能，也可以触发垃圾回收器，让它在一个特定的时间运行。可以使用 `collect()` 触发垃圾回收。

```
import gc
import pprint

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):
        print 'Linking nodes %s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

print

# Remove references to the graph nodes in this module's namespace
one = two = three = None

# Show the effect of garbage collection
for i in range(2):
    print 'Collecting %d ...' % i
    n = gc.collect()
    print 'Unreachable objects:', n
    print 'Remaining Garbage:',
    pprint.pprint(gc.garbage)
    print
```

在这个例子中，回收第一次运行时就会清除循环，因为除了自身以外，`Graph` 节点不再有其他引用。`collect()` 会返回它找到的“不可到达的”对象数。在这里，这个值是 6，因为共有 3 个对象，它们分别有自己的实例属性字典。

```
$ python gc_collect.py
```

```
Linking nodes Graph(one).next = Graph(two)
```

```
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)
```

```
Collecting 0 ...
Unreachable objects: 6
Remaining Garbage:[]
```

```
Collecting 1 ...
Unreachable objects: 0
Remaining Garbage:[]
```

不过，如果 Graph 有一个 `__del__()` 方法，垃圾回收器就无法断开循环。

```
import gc
import pprint

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):
        print '%s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)
    def __del__(self):
        print '%s.__del__()' % self

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

# Remove references to the graph nodes in this module's namespace
one = two = three = None
# Show the effect of garbage collection
print 'Collecting...'
n = gc.collect()
print 'Unreachable objects:', n
print 'Remaining Garbage:',
pprint.pprint(gc.garbage)
```

因为循环中不只一个对象有最终化方法，无法确定对象按什么顺序得到最终化处理以及垃圾回收。垃圾回收器主张力保安全，所以会保留对象。

```
$ python gc_collect_with_del.py
```

```

Graph(one).next = Graph(two)
Graph(two).next = Graph(three)
Graph(three).next = Graph(one)
Collecting...
Unreachable objects: 6
Remaining Garbage:[Graph(one), Graph(two), Graph(three)]

```

循环断开时，才能回收 Graph 实例。

```

import gc
import pprint

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):
        print 'Linking nodes %s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)
    def __del__(self):
        print '%s.__del__()' % self

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

# Remove references to the graph nodes in this module's namespace
one = two = three = None

# Collecting now keeps the objects as uncollectable
print
print 'Collecting...'
n = gc.collect()
print 'Unreachable objects:', n
print 'Remaining Garbage:',
pprint.pprint(gc.garbage)

# Break the cycle
print
print 'Breaking the cycle'
gc.garbage[0].set_next(None)
print 'Removing references in gc.garbage'

```

```

del gc.garbage[:]

# Now the objects are removed
print
print 'Collecting...'
n = gc.collect()
print 'Unreachable objects:', n
print 'Remaining Garbage:',
pprint.pprint(gc.garbage)

```

由于 `gc.garbage` 包含由之前运行垃圾回收得到的对象引用，因此需要在循环中断后进行清理，减少引用数，使对象能够得到最终化处理并释放。

```
$ python gc_collect_break_cycle.py
```

```

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

Collecting...
Unreachable objects: 6
Remaining Garbage:[Graph(one), Graph(two), Graph(three)]
Breaking the cycle
Linking nodes Graph(one).next = None
Removing references in gc.garbage
Graph(two).__del__()
Graph(three).__del__()
Graph(one).__del__()

Collecting...
Unreachable objects: 0
Remaining Garbage:[]

```

17.6.3 查找无法收集的对象引用

要在垃圾列表中查找哪个对象包含某个引用，这比查看一个对象引用了什么要麻烦一些。因为查找引用的代码本身也需要包含引用，因此需要忽略一些包含引用的对象（引用者）。下面这个例子创建了一个图循环，然后处理 `Graph` 实例，删除“父”节点中的引用。

```

import gc
import pprint
import Queue

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):

```



```

        print 'Linking nodes %s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)
    def __del__(self):
        print '%s.__del__()' % self

# Construct two graph cycles
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)
# Remove references to the graph nodes in this module's namespace
one = two = three = None

# Collecting now keeps the objects as uncollectable
print
print 'Collecting...'
n = gc.collect()
print 'Unreachable objects:', n
print 'Remaining Garbage:',
pprint.pprint(gc.garbage)

REFERRERS_TO_IGNORE = [ locals(), globals(), gc.garbage ]

def find_referring_graphs(obj):
    print 'Looking for references to %s' % repr(obj)
    referrers = (r for r in gc.get_referrers(obj)
                  if r not in REFERRERS_TO_IGNORE)
    for ref in referrers:
        if isinstance(ref, Graph):
            # A graph node
            yield ref
        elif isinstance(ref, dict):
            # An instance or other namespace dictionary
            for parent in find_referring_graphs(ref):
                yield parent

# Look for objects that refer to the objects that remain in
# gc.garbage.
print
print 'Clearing referrers:'
for obj in gc.garbage:
    for ref in find_referring_graphs(obj):
        ref.set_next(None)

```

```

    del ref # remove local reference so the node can be deleted
    del obj # remove local reference so the node can be deleted

# Clear references held by gc.garbage
print
print 'Clearing gc.garbage:'
del gc.garbage[:]

# Everything should have been freed this time
print
print 'Collecting...'
n = gc.collect()
print 'Unreachable objects:', n
print 'Remaining Garbage:',
pprint.pprint(gc.garbage)

```

如果循环是已知的，这种逻辑会有些“大材小用”，不过对于数据中一个不可解释的循环，使用 `get_referrers()` 可以暴露出预料之外的关系。

```
$ python gc_get_referrers.py
```

```

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

```

```

Collecting...
Unreachable objects: 6
Remaining Garbage:[Graph(one), Graph(two), Graph(three)]

```

```

Clearing referrers:
Looking for references to Graph(one)
Looking for references to {'name': 'three', 'next': Graph(one)}
Linking nodes Graph(three).next = None
Looking for references to Graph(two)
Looking for references to {'name': 'one', 'next': Graph(two)}
Linking nodes Graph(one).next = None
Looking for references to Graph(three)
Looking for references to {'name': 'two', 'next': Graph(three)}
Linking nodes Graph(two).next = None

```

```

Clearing gc.garbage:
Graph(three).__del__()
Graph(two).__del__()
Graph(one).__del__()

```

```

Collecting...
Unreachable objects: 0
Remaining Garbage:[]

```

17.6.4 回收阈限和代

垃圾回收器会维护它运行时看到的 3 个对象列表，分别对应回收器跟踪的各“代”（generation）。在各代中检查对象时，这些对象要么回收，要么变老进入下一代，直至最终达到永久保存的阶段。

根据回收器运行之间对象分配和撤销数之差，可以调整回收器例程以不同频率发生。如果分配数减去撤销数大于当前这一代的阈限，则运行垃圾回收器。当前阈限可以用 `gc_threshold()` 检查。

```
import gc
```

```
print gc.get_threshold()
```

返回值是一个元组，包含各代的阈限。

```
$ python gc_get_threshold.py
```

```
(700, 10, 10)
```

可以用 `set_threshold()` 改变阈限。下面这个示例程序从命令行读取第 0 代的阈限，调整 `gc` 设置，然后分配一系列对象。

```
import gc
import pprint
import sys
```

```
try:
```

```
    threshold = int(sys.argv[1])
```

```
except (IndexError, ValueError, TypeError):
```

```
    print 'Missing or invalid threshold, using default'
```

```
    threshold = 5
```

```
class MyObj(object):
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
        print 'Created', self.name
```

```
gc.set_debug(gc.DEBUG_STATS)
```

```
gc.set_threshold(threshold, 1, 1)
```

```
print 'Thresholds:', gc.get_threshold()
```

```
print 'Clear the collector by forcing a run'
```

```
gc.collect()
```

```
print
```

```
print 'Creating objects'
```

```
objs = []
```

```
for i in range(10):
```

```
    objs.append(MyObj(i))
```



不同阈值会导致在不同的时间完成垃圾回收清扫，如下所示（因为启用了调试）。

```
$ python -u gc_threshold.py 5

Thresholds: (5, 1, 1)
Clear the collector by forcing a run
gc: collecting generation 2...
gc: objects in each generation: 218 2683 0
gc: done, 0.0008s elapsed.

Creating objects
gc: collecting generation 0...
gc: objects in each generation: 7 0 2819
gc: done, 0.0000s elapsed.
Created 0
Created 1
Created 2
Created 3
Created 4
gc: collecting generation 0...
gc: objects in each generation: 6 4 2819
gc: done, 0.0000s elapsed.
Created 5
Created 6
Created 7
Created 8
Created 9
gc: collecting generation 2...
gc: objects in each generation: 5 6 2817
gc: done, 0.0007s elapsed.
```

较小的阈值会导致更频繁地运行清扫。

```
$ python -u gc_threshold.py 2
Thresholds: (2, 1, 1)
Clear the collector by forcing a run
gc: collecting generation 2...
gc: objects in each generation: 218 2683 0
gc: done, 0.0008s elapsed.

Creating objects
gc: collecting generation 0...
gc: objects in each generation: 3 0 2819
gc: done, 0.0000s elapsed.
gc: collecting generation 0...
gc: objects in each generation: 4 3 2819
gc: done, 0.0000s elapsed.
Created 0
Created 1
```



```

gc: collecting generation 1...
gc: objects in each generation: 3 4 2819
gc: done, 0.0000s elapsed.
Created 2
Created 3
Created 4
gc: collecting generation 0...
gc: objects in each generation: 5 0 2824
gc: done, 0.0000s elapsed.
Created 5
Created 6
Created 7
gc: collecting generation 0...
gc: objects in each generation: 5 3 2824
gc: done, 0.0000s elapsed.
Created 8
Created 9
gc: collecting generation 2...
gc: objects in each generation: 2 6 2820
gc: done, 0.0008s elapsed.

```

17.6.5 调试

调试内存泄漏可能很有难度。gc 包括有一些选项，可以提供内部状态，使这个任务更为容易一些。这些选项都是位标志，可以组合传递到 `set_debug()`，在程序运行时配置垃圾回收器。调试信息打印到 `sys.stderr`。

`DEBUG_STATS` 标志打开统计报告。这会使垃圾回收器报告每一代跟踪的对象数和完成清扫花费的时间。

```

import gc

gc.set_debug(gc.DEBUG_STATS)

gc.collect()

```

这个示例输出显示出运行了两次回收器。第一次是显式调用时运行，第二次是在解释器退出时运行。

```

$ python gc_debug_stats.py

gc: collecting generation 2...
gc: objects in each generation: 83 2683 0
gc: done, 0.0010s elapsed.
gc: collecting generation 2...
gc: objects in each generation: 0 0 2747
gc: done, 0.0008s elapsed.

```

启用 `DEBUG_COLLECTABLE` 和 `DEBUG_UNCOLLECTABLE` 会让回收器报告它检查的各个

对象能还是不能回收。这些标志需要与 `DEBUG_OBJECTS` 结合，以便 `gc` 打印所包含对象的信息。

```
import gc

flags = (gc.DEBUG_COLLECTABLE |
         gc.DEBUG_UNCOLLECTABLE |
         gc.DEBUG_OBJECTS
        )
gc.set_debug(flags)

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
        print 'Creating %s 0x%x (%s)' % \
            (self.__class__.__name__, id(self), name)
    def set_next(self, next):
        print 'Linking nodes %s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)

class CleanupGraph(Graph):
    def __del__(self):
        print '%s.__del__()' % self

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Construct another node that stands on its own
three = CleanupGraph('three')

# Construct a graph cycle with a finalizer
four = CleanupGraph('four')
five = CleanupGraph('five')
four.set_next(five)
five.set_next(four)

# Remove references to the graph nodes in this module's namespace
one = two = three = four = five = None

print

# Force a sweep
print 'Collecting'
```

```
gc.collect()
print 'Done'
```

这里构造了两个类 `Graph` 和 `CleanupGraph`，从而可以创建能够自动回收的结构，以及需要由用户显式中断循环的结构。

从输出可以看到，`Graph` 实例 `one` 和 `two` 创建了一个循环，不过仍能回收，因为它们没有终止器，而且它们惟一的进入引用来自其他可以回收的对象。尽管 `CleanupGraph` 有一个终止器，一旦 `three` 的引用数减至 0 就会被回收。与之相反，`four` 和 `five` 创建了一个循环，无法释放。

```
$ python -u gc_debug_collectable_objects.py

Creating Graph 0x100d99ad0 (one)
Creating Graph 0x100d99b10 (two)
Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(one)
Creating CleanupGraph 0x100d99b50 (three)
Creating CleanupGraph 0x100d99b90 (four)
Creating CleanupGraph 0x100d99bd0 (five)
Linking nodes CleanupGraph(four).next = CleanupGraph(five)
Linking nodes CleanupGraph(five).next = CleanupGraph(four)
CleanupGraph(three).__del__()

Collecting
gc: collectable <Graph 0x100d99ad0>
gc: collectable <Graph 0x100d99b10>
gc: collectable <dict 0x100c5b8e0>
gc: collectable <dict 0x100c5cb70>
gc: uncollectable <CleanupGraph 0x100d99b90>
gc: uncollectable <CleanupGraph 0x100d99bd0>
gc: uncollectable <dict 0x100c5cc90>
gc: uncollectable <dict 0x100c5cff0>
Done
```

对于老式类（并非派生自 `object`），标志 `DEBUG_INSTANCES` 的做法基本相同。

```
import gc

flags = (gc.DEBUG_COLLECTABLE |
         gc.DEBUG_UNCOLLECTABLE |
         gc.DEBUG_INSTANCES
        )
gc.set_debug(flags)

class Graph:
    def __init__(self, name):
        self.name = name
        self.next = None
```

```

        print 'Creating %s 0x%x (%s)' % \
            (self.__class__.__name__, id(self), name)
    def set_next(self, next):
        print 'Linking nodes %s.next = %s' % (self, next)
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)

class CleanupGraph(Graph):
    def __del__(self):
        print '%s.__del__()' % self

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Construct another node that stands on its own
three = CleanupGraph('three')

# Construct a graph cycle with a finalizer
four = CleanupGraph('four')
five = CleanupGraph('five')
four.set_next(five)
five.set_next(four)

# Remove references to the graph nodes in this module's namespace
one = two = three = four = five = None

print

# Force a sweep
print 'Collecting'
gc.collect()
print 'Done'

```

不过, 在这种情况下, 保存实例属性的 dict 对象不包含在输出中。

```
$ python -u gc_debug_collectable_instances.py
```

```

Creating Graph 0x100da23f8 (one)
Creating Graph 0x100da2440 (two)
Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(one)
Creating CleanupGraph 0x100da24d0 (three)
Creating CleanupGraph 0x100da2518 (four)
Creating CleanupGraph 0x100da2560 (five)

```




```

Linking nodes CleanupGraph(four).next = CleanupGraph(five)
Linking nodes CleanupGraph(five).next = CleanupGraph(four)
CleanupGraph(three).__del__()

```

```

Collecting
gc: collectable <Graph instance at 0x100da23f8>
gc: collectable <Graph instance at 0x100da2440>
gc: uncollectable <CleanupGraph instance at 0x100da2518>
gc: uncollectable <CleanupGraph instance at 0x100da2560>
Done

```

如果可以看到无法收集的对象，但这还不能提供足够的信息来了解数据保留在哪里，可以启用 `DEBUG_SAVEALL`，让 `gc` 保留它找到的所有对象，而在 `garbage` 列表中没有任何引用。

```

import gc

flags = (gc.DEBUG_COLLECTABLE |
         gc.DEBUG_UNCOLLECTABLE |
         gc.DEBUG_OBJECTS |
         gc.DEBUG_SAVEALL
        )

gc.set_debug(flags)

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):
        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)

class CleanupGraph(Graph):
    def __del__(self):
        print '%s.__del__()' % self

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Construct another node that stands on its own
three = CleanupGraph('three')

# Construct a graph cycle with a finalizer
four = CleanupGraph('four')
five = CleanupGraph('five')

```

```

four.set_next(five)
five.set_next(four)

# Remove references to the graph nodes in this module's namespace
one = two = three = four = five = None

# Force a sweep
print 'Collecting'
gc.collect()
print 'Done'

# Report on what was left
for o in gc.garbage:
    if isinstance(o, Graph):
        print 'Retained: %s 0x%x' % (o, id(o))

```

这样就允许在垃圾回收之后还能检查对象，这对于某些情况很有帮助，例如，如果构造函数不能修改为在创建各个对象时打印对象 id。

```
$ python -u gc_debug_saveall.py
```

```

CleanupGraph(three).__del__()
Collecting
gc: collectable <Graph 0x100d99b10>
gc: collectable <Graph 0x100d99b50>
gc: collectable <dict 0x100c5c740>
gc: collectable <dict 0x100c5cb60>
gc: uncollectable <CleanupGraph 0x100d99bd0>
gc: uncollectable <CleanupGraph 0x100d99c10>
gc: uncollectable <dict 0x100c5cc80>
gc: uncollectable <dict 0x100c5cfe0>
Done
Retained: Graph(one) 0x100d99b10
Retained: Graph(two) 0x100d99b50
Retained: CleanupGraph(four) 0x100d99bd0
Retained: CleanupGraph(five) 0x100d99c10

```

为简单起见，DEBUG_LEAK 定义为所有其他选项的一个组合。

```

import gc

flags = gc.DEBUG_LEAK

gc.set_debug(flags)

class Graph(object):
    def __init__(self, name):
        self.name = name
        self.next = None
    def set_next(self, next):

```



```

        self.next = next
    def __repr__(self):
        return '%s(%s)' % (self.__class__.__name__, self.name)

class CleanupGraph(Graph):
    def __del__(self):
        print '%s.__del__()' % self

# Construct a graph cycle
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Construct another node that stands on its own
three = CleanupGraph('three')

# Construct a graph cycle with a finalizer
four = CleanupGraph('four')
five = CleanupGraph('five')
four.set_next(five)
five.set_next(four)

# Remove references to the graph nodes in this module's namespace
one = two = three = four = five = None

# Force a sweep
print 'Collecting'
gc.collect()
print 'Done'

# Report on what was left
for o in gc.garbage:
    if isinstance(o, Graph):
        print 'Retained: %s 0x%x' % (o, id(o))

```

要记住，由于 `DEBUG_SAVEALL` 由 `DEBUG_LEAK` 启用，甚至正常情况下已经被回收和删除的无引用的对象也会保留。

```
$ python -u gc_debug_leak.py
```

```

CleanupGraph(three).__del__()
Collecting
gc: collectable <Graph 0x100d99b10>
gc: collectable <Graph 0x100d99b50>
gc: collectable <dict 0x100c5b8d0>
gc: collectable <dict 0x100c5cad0>
gc: uncollectable <CleanupGraph 0x100d99bd0>

```

```
gc: uncollectable <CleanupGraph 0x100d99c10>
gc: uncollectable <dict 0x100c5cbf0>
gc: uncollectable <dict 0x100c5cf50>
Done
Retained: Graph(one) 0x100d99b10
Retained: Graph(two) 0x100d99b50
Retained: CleanupGraph(four) 0x100d99bd0
Retained: CleanupGraph(five) 0x100d99c10
```

参见:

gc (<http://docs.python.org/library/gc.html>) 这个模块的标准库文档。

weakref (2.7 节) weakref 模块提供了一种创建对象引用的方法, 而不会增加其引用数, 使对象仍能得到垃圾回收。

Supporting Cyclic Garbage Collection (<http://docs.python.org/c-api/gcsupport.html>) Python C API 文档的背景资料。

How does Python manage memory? (<http://effbot.org/pyfaq/how-does-pythonmanage-memory.htm>) Fredrik Lundh 撰写的关于 Python 内存管理的一篇文章。

17.7 sysconfig——解释器编译时配置

作用: 访问用于构建 Python 的配置设置。

Python 版本: 2.7 及以后版本

在 Python 2.7 中, sysconfig 已经从 distutils 抽取出来, 成为一个独立的模块。它包括一些函数来确定用来编译和安装当前解释器的设置。

17.7.1 配置变量

可以通过两个函数访问构建时配置设置。get_config_vars() 返回一个字典, 将配置变量名映射到值。

```
import sysconfig

config_values = sysconfig.get_config_vars()
print 'Found %d configuration settings' % len(config_values.keys())
print

print 'Some highlights:'

print
print '  Installation prefixes:'
print '    prefix={prefix}'.format(**config_values)
print '    exec_prefix={exec_prefix}'.format(**config_values)

print
```

```

print ' Version info:'
print '     py_version={py_version}'.format(**config_values)
print '     py_version_short={py_version_short}'.format(**config_values)
print '     py_version_nodot={py_version_nodot}'.format(**config_values)

print
print ' Base directories:'
print '     base={base}'.format(**config_values)
print '     platbase={platbase}'.format(**config_values)
print '     userbase={userbase}'.format(**config_values)
print '     srcdir={srcdir}'.format(**config_values)

print
print ' Compiler and linker flags:'
print '     LDFLAGS={LDFLAGS}'.format(**config_values)
print '     BASECFLAGS={BASECFLAGS}'.format(**config_values)
print '     Py_ENABLE_SHARED={Py_ENABLE_SHARED}'.format(**config_values)

```

sysconfig API 提供的详细级别取决于运行程序所在的平台。在 POSIX 系统（如 Linux 和 OS X）上，用来构建解释器的 Makefile 以及为构建生成的 config.h 头文件都会得到解析，其中找到的所有变量都可用。在非 POSIX 系统（如 Windows）上，设置则仅限于一些路径、文件名扩展和版本详细信息。

```
$ python sysconfig_get_config_vars.py
```

```
Found 511 configuration settings
```

```
Some highlights:
```

```
Installation prefixes:
```

```
prefix=/Library/Frameworks/Python.framework/Versions/2.7
exec_prefix=/Library/Frameworks/Python.framework/Versions/2.7
```

```
Version info:
```

```
py_version=2.7
py_version_short=2.7
py_version_nodot=27
```

```
Base directories:
```

```
base=/Users/dhellmann/.virtualenvs/pymotw
platbase=/Users/dhellmann/.virtualenvs/pymotw
userbase=/Users/dhellmann/Library/Python/2.7
srcdir=/Users/sysadmin/X/r27
```

```
Compiler and linker flags:
```

```
LDFLAGS=-arch i386 -arch ppc -arch x86_64 -isysroot / -g
BASECFLAGS=-fno-strict-aliasing -fno-common -dynamic
Py_ENABLE_SHARED=0
```



向 `get_config_vars()` 传递变量名，会把返回值改为一个 list，这是将所有这些变量追加在一起创建的。

```
import sysconfig

bases = sysconfig.get_config_vars('base', 'platbase', 'userbase')
print 'Base directories:'
for b in bases:
    print ' ', b
```

这个例子建立了所有安装基目录（可以在当前系统上找到模块）的一个列表。

```
$ python sysconfig_get_config_vars_by_name.py
```

```
Base directories:
/Users/dhellmann/.virtualenvs/pymotw
/Users/dhellmann/.virtualenvs/pymotw
/Users/dhellmann/Library/Python/2.7
```

只需要一个配置值时，可以使用 `get_config_var()` 来获取。

```
import sysconfig

print 'User base directory:', sysconfig.get_config_var('userbase')
print 'Unknown variable   :', sysconfig.get_config_var('NoSuchVariable')
```

如果变量没有找到，`get_config_var()` 会返回 `None` 而不是产生一个异常。

```
$ python sysconfig_get_config_var.py
```

```
User base directory: /Users/dhellmann/Library/Python/2.7
Unknown variable   : None
```

17.7.2 安装路径

`sysconfig` 主要由安装和打包工具使用。因此，尽管可以用来访问通用的配置设置（如解释器版本），但它主要用来访问另外一些信息，即查找一个系统上当前安装的 Python 发布中各个部分所在位置所需的信息。安装一个包所用的位置依赖于使用的方案（scheme）。

方案是一组平台特定的默认目录，根据平台的打包标准和原则来组织。安装到一个站点范围位置或是用户所有的一个私有目录时，分别有不同的方案。可以用 `get_scheme_names()` 访问完整的方案集。

```
import sysconfig

for name in sysconfig.get_scheme_names():
    print name
```

这里没有“当前方案”的概念。默认方案取决于平台，使用的具体方案依赖于为安装程序提供的选项。如果当前系统在运行一个遵循 POSIX 的操作系统，则默认方案为 `posix_prefix`。否则，按照 `os.name` 的定义，默认为操作系统名。

```
$ python sysconfig_get_scheme_names.py
```

```
nt
nt_user
os2
os2_home
osx_framework_user
posix_home
posix_prefix
posix_user
```

每个方案定义了一组用于安装包的路径。要得到路径名列表，可以使用 `get_path_names()`。

```
import sysconfig

for name in sysconfig.get_path_names():
    print name
```

对于一个给定方案，有些路径可能是相同的，不过安装工具对于哪些是真正的路径不能做任何假设。每个名都有一个特定的语义含义，所以应当在安装期间用正确的名字来查找一个给定文件的路径。路径名及其含义的完整列表见表 17.4。

表 17.4 sysconfig 中使用的路径名

路 径 名	描 述
stdlib	标准 Python 库文件，非平台特定
platstdlib	标准 Python 库文件，平台特定
platlib	站点特定、平台特定文件
purelib	站点特定、非平台特定文件
include	头文件，非平台特定
platinclude	头文件，平台特定
scripts	可执行脚本文件
data	数据文件

```
$ python sysconfig_get_path_names.py
```

```
stdlib
platstdlib
purelib
platlib
include
scripts
data
```

使用 `get_paths()` 可以获取与一个方案关联的具体目录。

```
import sysconfig
import pprint
```

```

import os
for scheme in ['posix_prefix', 'posix_user']:
    print scheme
    print '=' * len(scheme)
    paths = sysconfig.get_paths(scheme=scheme)
    prefix = os.path.commonprefix(paths.values())
    print 'prefix = %s\n' % prefix
    for name, path in sorted(paths.items()):
        print '%s\n .%s' % (name, path[len(prefix):])
    print

```

这个例子显示了对应 `posix_prefix` 的系统范围路径（在 Mac OS X 上构建的一个框架之下）和对应 `posix_user` 的用户特定值之间的差别。

```
$ python sysconfig_get_paths.py
```

```

posix_prefix
=====
prefix = /Library/Frameworks/Python.framework/Versions/2.7

data

include
    ./include/python2.7
platinclude
    ./include/python2.7
platlib
    ./lib/python2.7/site-packages
platstdlib
    ./lib/python2.7
purelib
    ./lib/python2.7/site-packages
scripts
    ./bin
stdlib
    ./lib/python2.7

posix_user
=====
prefix = /Users/dhellmann/Library/Python/2.7

data
.
include
    ./include/python2.7
platlib
    ./lib/python2.7/site-packages
platstdlib

```




```

./lib/python2.7
purelib
./lib/python2.7/site-packages
scripts
./bin
stdlib
./lib/python2.7

```

要得到单个路径，可以调用 `get_path()`。

```

import sysconfig
import pprint

for scheme in ['posix_prefix', 'posix_user']:
    print scheme
    print '=' * len(scheme)
    print 'purelib =', sysconfig.get_path(name='purelib',
                                          scheme=scheme)
    print

```

使用 `get_path()` 等价于保存 `get_paths()` 的值，并在字典中查找单个键。如果需要多个路径，`get_paths()` 更为高效，因为它不会每次重新计算所有路径。

```
$ python sysconfig_get_path.py
```

```

posix_prefix
=====
purelib = /Library/Frameworks/Python.framework/Versions/2.7/site-\
packages

posix_user
=====
purelib = /Users/dhellmann/Library/Python/2.7/lib/python2.7/site-\
packages

```

17.7.3 Python 版本和平台

尽管 `sys` 包含一些基本平台标识，但还不够特定，不足以用来安装二进制包，因为 `sys.platform` 并不总包括有关硬件体系结构、指令大小或影响二进制库兼容性的其他值的信息。要想更准确地指示平台信息，可以使用 `get_platform()`。

```

import sysconfig

print sysconfig.get_platform()

```

尽管这个示例输出在一个 OS X 10.6 系统上得到，但编译解释器时保证了与 10.5 的兼容性，所以平台串中包含了这个版本号 (10.5)。

```
$ python sysconfig_get_platform.py
```

macosx-10.5-fat3

作为一种便利方法, 还可以通过 `sysconfig` 中的 `get_python_version()` 由 `sys.version_info` 得到解释器版本。

```
import sysconfig
import sys

print 'sysconfig.get_python_version():', sysconfig.get_python_version()
print '\nsys.version_info:'
print '  major      :', sys.version_info.major
print '  minor      :', sys.version_info.minor
print '  micro      :', sys.version_info.micro
print '  releaselevel:', sys.version_info.releaselevel
print '  serial     :', sys.version_info.serial
```

`get_python_version()` 会返回一个串, 构建版本特定的路径时很适用。

```
$ python sysconfig_get_python_version.py
```

```
sysconfig.get_python_version(): 2.7
sys.version_info:
  major      : 2
  minor      : 7
  micro      : 0
  releaselevel: final
  serial     : 0
```

参见:

`sysconfig` (<http://docs.python.org/library/sysconfig.html>) 这个模块的标准库文档。

`distutils` `sysconfig` 用作为 `distutils` 包的一部分。

`distutils2` (<http://hg.python.org/distutils2/>) 更新到 `distutils`, 由 Tarek Ziade 管理。

`site` (17.1 节) `site` 模块更详细地描述了导入时搜索的路径。

`os` (17.3 节) 包括 `os.name`, 当前操作系统的名字。

`sys` (17.2 节) 包括其他构建时信息, 如平台。



第 18 章

语言工具

除了上一章介绍的开发工具之外，Python 还包括一些模块，允许访问其内部特性。本章不分应用领域来介绍 Python 中使用的一些工具。

warnings 模块用于报告非致命条件或可恢复的错误。如果标准库的一个特性被一个新类、接口或模块取代，会生成 DeprecationWarning，这就是警告的一个常见例子。使用 warnings 会报告可能需要用户注意的条件，但是不是致命的。

定义符合一个公共 API 的一组类时，如果这个 API 由另外某个人定义或者使用了大量方法，这可能很有难度。要解决这个问题，一种常用方法是从一个公共基类派生所有新类。不过，哪些方法应当覆盖而哪些可以采用默认行为，这一点并不总是显而易见。abc 模块的抽象基类提供了 API 的形式化定义，显式地标志某些方法必须由类以某种方式提供，从而避免在类未完全实现时就进行实例化。例如，Python 的很多容器类型都在 abc 或 collections 中定义了抽象基类。

dis 模块可以用于反汇编程序的字节码版本，来了解解释器运行这个程序时所采取的步骤。调试性能问题或并发问题时，查看反汇编的代码可能很有用，因为它能给出解释器为程序中的各个语句执行的原子操作。

inspect 模块为当前进程中的所有对象提供了自省支持。这包括导入的模块、类和函数定义，以及由这些定义实例化的“现场”对象。可以利用自省为源代码生成文档、动态地调整运行时行为，或者检查一个程序的执行环境。

exceptions 模块定义了整个标准库和第三方模块中使用的公共异常。如果熟悉异常的类层次结构，就能更容易地理解错误消息，并创建可以适当处理异常的健壮代码。

18.1 warnings——非致命警告

作用：向用户提供非致命警告，指出运行一个程序时遇到的问题。

Python 版本：2.1 及以后版本

warnings 模块由 PEP 230 引入，因为预期到 Python 3.0 中会出现不能向后兼容的改变，所以以这种方式来警告程序员，指出语言或库特性已经改变。这个模块还可以用来报告可恢复的配置错误或者因为缺少库而出现的特性降级。不过，最好通过 logging 模块提供用户可见的消息，因为发送到控制台的警告可能丢失。

由于警告不是致命的，因此程序在运行过程中可能会多次遇到相同的警告情况。warnings

模块会抑制相同来源的重复消息，避免因为反复看到同样的警告而厌烦。可以逐情况地控制输出，为此可以使用解释器的命令行选项，也可以调用 `warnings` 中的函数。

18.1.1 分类和过滤

警告使用内置异常类 `Warning` 的子类进行分类。`exceptions` 模块的联机文档中描述了很多标准值，还可以通过派生 `Warning` 增加定制警告。

警告要根据过滤器 (filter) 设置来处理。过滤器包括 5 个部分：动作 (action)、消息 (message)、类别 (category)、模块 (module) 和行号 (line number)。过滤器的消息部分是一个正则表达式，用来匹配警告文本。类别是一个异常类的类名。模块包含一个正则表达式，要与生成警告的模块名匹配。行号可以用来改变在一个警告具体出现时的处理。

生成一个警告时，要把它与所有注册的过滤器比较。第一个匹配的过滤器将控制对这个警告采取的动作。如果没有匹配的过滤器，则会采取默认动作。过滤机制理解的动作如表 18.1 所示。

表 18.1 警告过滤器动作

动 作	含 义
<code>error</code>	将警告变成一个异常
<code>ignore</code>	删除警告
<code>always</code>	总是发出一个警告
<code>default</code>	从各个位置第一次生成警告时输出警告
<code>module</code>	从各个模块第一次生成警告时输出警告
<code>once</code>	第一次生成警告时输出警告

18.1.2 生成警告

要发出一个警告，最简单的方法是调用 `warn()`，并提供消息作为参数。

```
import warnings
```

```
print 'Before the warning'
warnings.warn('This is a warning message')
print 'After the warning'
```

程序运行时，会输出这条消息。

```
$ python -u warnings_warn.py
```

```
Before the warning
warnings_warn.py:13: UserWarning: This is a warning message
  warnings.warn('This is a warning message')
After the warning
```

尽管输出了警告，但默认行为是经过这一点继续运行余下的程序。可以用一个过滤器改变

这个行为。

```
import warnings

warnings.simplefilter('error', UserWarning)

print 'Before the warning'
warnings.warn('This is a warning message')
print 'After the warning'
```

在这个例子中，`simplefilter()` 函数为内部过滤器列表增加了一项，告诉 `warnings` 模块出现一个 `UserWarning` 警告时要产生一个异常。

```
$ python -u warnings_warn_raise.py

Before the warning
Traceback (most recent call last):
  File "warnings_warn_raise.py", line 15, in <module>
    warnings.warn('This is a warning message')
UserWarning: This is a warning message
```

也可以使用解释器的 `-W` 选项从命令行控制过滤器行为。只需指定过滤器属性，这是一个包含 5 部分的串（动作、消息、类别、模块和行号），各部分之间用冒号（:）分隔。例如，如果运行 `warnings_warn.py`，而且过滤器设置为在出现 `UserWarning` 时产生一个错误，则会生成一个异常。

```
$ python -u -W "error::UserWarning::0" warnings_warn.py

Before the warning
Traceback (most recent call last):
  File "warnings_warn.py", line 13, in <module>
    warnings.warn('This is a warning message')
UserWarning: This is a warning message
```

由于消息和模块字段为空，这解释为与所有内容匹配。

18.1.3 用模式过滤

要通过编程按照更复杂的规则进行过滤，可以使用 `filterwarnings()`。例如，要根据消息文本的内容过滤，可以提供正则表达式作为消息参数。

```
import warnings

warnings.filterwarnings('ignore', '.*do not.*',)

warnings.warn('Show this message')
warnings.warn('Do not show this message')
```

模式包含“do not”，具体的消息使用了“Do not”。这个模式会匹配，因为正则表达式会编译为查找不区分大小写的匹配。

```
$ python warnings_filterwarnings_message.py
```

```
warnings_filterwarnings_message.py:14: UserWarning: Show this message
  warnings.warn('Show this message')
```

示例程序 `warnings_filtering.py` 会生成两个警告。

```
import warnings
```

```
warnings.warn('Show this message')
warnings.warn('Do not show this message')
```

在命令行上使用过滤器参数，可以忽略其中一个警告。

```
$ python -W "ignore:do not:UserWarning::0" warnings_filtering.py
```

```
warnings_filtering.py:12: UserWarning: Show this message
  warnings.warn('Show this message')
```

同样的模式匹配规则也应用于源模块名（该模块包含生成警告的调用）。可以将模块名作为模式传至 `module` 参数，抑制来自 `warnings_filtering` 模块的所有消息。

```
import warnings
```

```
warnings.filterwarnings('ignore',
                        '.*',
                        UserWarning,
                        'warnings_filtering',
                        )
```

```
import warnings_filtering
```

由于有这个过滤器，导入 `warnings_filtering` 时不会发出任何警告。

```
$ python warnings_filterwarnings_module.py
```

如果只抑制 `warnings_filtering` 第 13 行上的消息，可以将这个行号作为 `filterwarnings()` 的最后一个参数。可以使用源文件中的实际行号来限制过滤器，也可以使用 0，对消息的所有出现都应用过滤器。

```
import warnings
```

```
warnings.filterwarnings('ignore',
                        '.*',
                        UserWarning,
                        'warnings_filtering',
                        13)
```

```
import warnings_filtering
```

这个模式与所有消息都匹配，所以重要的参数是模块名和行号。

```
$ python warnings_filterwarnings_lineno.py

/Users/dhellmann/Documents/PyMOTW/book/PyMOTW/warnings/warnings_filter
ing.py:12: UserWarning: Show this message
  warnings.warn('Show this message')
```

18.1.4 重复的警告

默认情况下，大多数警告只是在一个给定位置第一次出现时才会输出，这里所说的“位置”要由模块和生成警告的相应行号组合定义。

```
import warnings

def function_with_warning():
    warnings.warn('This is a warning!')
```

```
function_with_warning()
function_with_warning()
function_with_warning() .
```

这个例子多次调用同一个函数，不过只生成一个警告。

```
$ python warnings_repeated.py
```

```
warnings_repeated.py:13: UserWarning: This is a warning!
  warnings.warn('This is a warning!')
```

“once”动作可以用来抑制相同消息在不同位置多次出现。

```
import warnings

warnings.simplefilter('once', UserWarning)
```

```
warnings.warn('This is a warning!')
warnings.warn('This is a warning!')
warnings.warn('This is a warning!')
```

所有警告的消息文本会保存，而且只输出一条消息。

```
$ python warnings_once.py
```

```
warnings_once.py:14: UserWarning: This is a warning!
  warnings.warn('This is a warning!')
```

类似地，“module”可以抑制来自相同模块的重复消息，而不论具体行号是什么。

18.1.5 候选消息传送函数

正常情况下，警告会输出到 `sys.stderr`。可以替换 `warnings` 模块中的 `showwarning()` 函数，来改变这个行为。例如，要把警告发送到一个日志文件而不是标准错误输出，可以把 `showwarning()` 替换为将该警告记入日志的一个函数。

```

import warnings
import logging
logging.basicConfig(level=logging.INFO)

def send_warnings_to_log(message, category, filename, lineno, file=None):
    logging.warning(
        '%s:%s: %s:%s' %
        (filename, lineno, category.__name__, message))
    return

old_showwarning = warnings.showwarning
warnings.showwarning = send_warnings_to_log

warnings.warn('message')

调用 warn() 时，警告会随其余日志消息发出。
$ python warnings_showwarning.py

WARNING:root:warnings_showwarning.py:24: UserWarning:message

```

18.1.6 格式化

如果警告要发送到标准错误输出，但是它需要重新格式化，可以替换 `formatwarning()`。

```

import warnings

def warning_on_one_line(message, category, filename, lineno,
                        file=None, line=None):
    return '-> %s:%s: %s:%s' % \
        (filename, lineno, category.__name__, message)

warnings.warn('Warning message, before')
warnings.formatwarning = warning_on_one_line
warnings.warn('Warning message, after')

格式化函数必须返回一个串，其中包含要向用户显示的警告表示。
$ python -u warnings_formatwarning.py

warnings_formatwarning.py:17: UserWarning: Warning message, before
warnings.warn('Warning message, before')
-> warnings_formatwarning.py:19: UserWarning:Warning message, after

```

18.1.7 警告中的栈层次

默认情况下，警告消息包括生成该消息的源代码行（如果有）。不过，只是看到具体警告消息和代码行并不太有用。实际上，可以告诉 `warn()` 要在栈中上行多远才能找到调用相应函数（即包含这个警告的函数）的代码行。这样一来，使用了已弃废函数的用户就可以看到函数在

哪里调用，而不是看到函数的实现。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import warnings
5
6  def old_function():
7      warnings.warn(
8          'old_function() is deprecated, use new_function() instead',
9          stacklevel=2)
10
11 def caller_of_old_function():
12     old_function()
13
14 caller_of_old_function()

```

在这个例子中，`warn()` 需要在栈中上行两层，一层对应它自身，另一层对应 `old_function()`。

```
$ python warnings_warn_stacklevel.py
```

```

warnings_warn_stacklevel.py:12: UserWarning: old_function() is
deprecated, use new_function() instead
  old_function()

```

参见：

`warnings` (<http://docs.python.org/lib/module-warnings.html>) 这个模块的标准库文档。

PEP 230 (www.python.org/dev/peps/pep-0230) 警告框架。

`exceptions` (18.5 节) 异常和警告的基类。

`logging` (14.9 节) 传送警告的一种候选机制是写入日志。

18.2 abc——抽象基类

作用：定义和使用抽象基类完成接口验证。

Python 版本：2.6 及以后版本

18.2.1 为什么使用抽象基类

抽象基类是一种接口，与单个 `hasattr()` 检查特定方法相比，抽象基类的检查更为严格。通过定义一个抽象基类，可以为一组子类建立一个公共 API。有些情况下，可能需要一个对应用源代码不太熟悉的人提供插件扩展，这种情况下这个功能就特别有用，另外对于大型团队合作或者处理一个很大的代码库（同时跟踪所有类很困难，甚至不可能）也很有帮助。

18.2.2 抽象基类如何工作

abc 的做法是，将基类的方法标志为抽象，然后注册具体类作为这个抽象基类的实现。如果应用或库需要一个特定的 API，可以用 `issubclass()` 或 `isinstance()` 根据抽象类检查对象。

首先，定义一个抽象基类表示一组插件的 API，用于保存和加载数据。设置新基类的 `__metaclass__` 为 `ABCMeta`，使用 `abstractmethod()` 装饰符为这个类建立公共 API。下面的例子使用了 `abc_base.py`，其中包含一组应用插件的基类。

```
import abc

class PluginBase(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source
        and return an object.
        """

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""
```

18.2.3 注册一个具体类

有两种方法指示一个具体类实现了一个抽象 API：或者显式地注册这个类，或者直接从抽象基类创建新的子类。当类提供了所需的 API 时，可以使用 `register()` 类方法显式地添加一个具体类，但是它不属于抽象基类的继承树。

```
import abc
from abc_base import PluginBase

class LocalBaseClass(object):
    pass

class RegisteredImplementation(LocalBaseClass):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)

PluginBase.register(RegisteredImplementation)

if __name__ == '__main__':
    print 'Subclass:', issubclass(RegisteredImplementation,
```



```

        PluginBase)
    print 'Instance:', isinstance(RegisteredImplementation(),
        PluginBase)

```

在这个例子中，RegisteredImplementation 派生自 LocalBaseClass，但是它注册为实现 PluginBase API。这说明 isinstance() 和 isinstance() 会把它看作是从 PluginBase 派生的。

```
$ python abc_register.py
```

```

Subclass: True
Instance: True

```

18.2.4 通过派生实现

直接从基类派生子类可以避免显式地注册类。

```

import abc
from abc_base import PluginBase

class SubclassImplementation(PluginBase):

    def load(self, input):
        return input.read()
    def save(self, output, data):
        return output.write(data)

if __name__ == '__main__':
    print 'Subclass:', isinstance(SubclassImplementation, PluginBase)
    print 'Instance:', isinstance(SubclassImplementation(), PluginBase)

```

在这里，使用了常规的 Python 类管理特性来识别 PluginImplementation 实现了抽象基类 PluginBase。

```
$ python abc_subclass.py
```

```

Subclass: True
Instance: True

```

使用直接派生有一个副作用：向基类询问由其派生类的列表时，可以找到一个插件的所有实现（这不是 abc 特有的一个特性，所有类都有这个特性）。

```

import abc
from abc_base import PluginBase
import abc_subclass
import abc_register

for sc in PluginBase.__subclasses__():
    print sc.__name__

```

尽管导入了 abc_register()，但 RegisteredImplementation 并不在子类列表中，因为它并非真

正派生自这个基类。

```
$ python abc_find_subclasses.py
```

```
SubclassImplementation
```

不完整的实现

直接从抽象基类派生子类还有一个好处，除非子类完全实现了 API 的抽象部分，否则子类不能实例化。

```
import abc
from abc_base import PluginBase
class IncompleteImplementation(PluginBase):

    def save(self, output, data):
        return output.write(data)

PluginBase.register(IncompleteImplementation)

if __name__ == '__main__':
    print 'Subclass:', isinstance(IncompleteImplementation,
                                   PluginBase)
    print 'Instance:', isinstance(IncompleteImplementation(),
                                   PluginBase)
```

这会避免不完整的实现在运行时触发未预期的错误。

```
$ python abc_incomplete.py
```

```
Subclass: True
```

```
Instance:
```

```
Traceback (most recent call last):
```

```
File "abc_incomplete.py", line 23, in <module>
```

```
    print 'Instance:', isinstance(IncompleteImplementation(),
```

```
TypeError: Can't instantiate abstract class
```

```
IncompleteImplementation with abstract methods load
```

18.2.5 abc 中的具体方法

具体类必须提供所有抽象方法的实现，不仅如此，抽象基类也可以提供实现，可以通过 `super()` 来调用。这就允许将公共逻辑放在基类中从而得到重用，而要求子类用（可能的）定制逻辑提供一个覆盖方法。

```
import abc
from cStringIO import StringIO

class ABCWithConcreteImplementation(object):
    __metaclass__ = abc.ABCMeta
```

```

@abc.abstractmethod
def retrieve_values(self, input):
    print 'base class reading data'
    return input.read()
class ConcreteOverride(ABCWithConcreteImplementation):

    def retrieve_values(self, input):
        base_data = super(ConcreteOverride,
                           self).retrieve_values(input)
        print 'subclass sorting data'
        response = sorted(base_data.splitlines())
        return response

input = StringIO("""line one
line two
line three
""")

reader = ConcreteOverride()
print reader.retrieve_values(input)
print

```

由于 `ABCWithConcreteImplementation()` 是一个抽象基类，因此不可能实例化这个类直接使用。子类必须为 `retrieve_values()` 提供一个覆盖方法，在这里，具体类在返回数据之前要先整理数据。

```

$ python abc_concrete_method.py

base class reading data
subclass sorting data
['line one', 'line three', 'line two']

```

18.2.6 抽象属性

如果一个 API 规范除了方法外还包括属性，可以用 `@abstractproperty` 定义来要求具体类中应包括这些属性。

```

import abc

class Base(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractproperty
    def value(self):
        return 'Should never get here'
    @abc.abstractproperty
    def constant(self):
        return 'Should never get here'

```

```

class Implementation(Base):
    @property
    def value(self):
        return 'concrete property'

    constant = 'set by a class attribute'

try:
    b = Base()
    print 'Base.value:', b.value
except Exception, err:
    print 'ERROR:', str(err)

i = Implementation()
print 'Implementation.value :', i.value
print 'Implementation.constant:', i.constant

```

这个例子中的 Base 类不能实例化，因为它对于 value 和 constant 只有一个抽象的属性获取方法。在 Implementation 中为 value 属性提供了一个具体的获取方法，另外使用一个类属性来定义 constant。

```
$ python abc_abstractproperty.py
```

```

ERROR: Can't instantiate abstract class Base with abstract
methods constant, value
Implementation.value : concrete property
Implementation.constant: set by a class attribute

```

还可以定义抽象的读写属性。

```

import abc

class Base(object):
    __metaclass__ = abc.ABCMeta

    def value_getter(self):
        return 'Should never see this'
    def value_setter(self, newvalue):
        return

    value = abc.abstractproperty(value_getter, value_setter)

class PartialImplementation(Base):
    @abc.abstractproperty
    def value(self):
        return 'Read-only'

class Implementation(Base):

```

```

    _value = 'Default value'

    def value_getter(self):
        return self._value

    def value_setter(self, newvalue):
        self._value = newvalue

    value = property(value_getter, value_setter)

try:
    b = Base()
    print 'Base.value:', b.value
except Exception, err:
    print 'ERROR:', str(err)

try:
    p = PartialImplementation()
    print 'PartialImplementation.value:', p.value
except Exception, err:
    print 'ERROR:', str(err)

i = Implementation()
print 'Implementation.value:', i.value

i.value = 'New value'
print 'Changed value:', i.value

```

与抽象属性一致，具体属性必须用同样的方式定义。如果试图用一个只读属性覆盖 PartialImplementation 中的一个读写属性，这是行不通的。

```
$ python abc_abstractproperty_rw.py
```

```

ERROR: Can't instantiate abstract class Base with abstract
methods value
ERROR: Can't instantiate abstract class PartialImplementation
with abstract methods value
Implementation.value: Default value
Changed value: New value

```

要对读写抽象属性使用装饰符语法，获取和设置值的方法必须同名。

```

import abc

class Base(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractproperty
    def value(self):

```

```

        return 'Should never see this'

    @value.setter
    def value(self, newvalue):
        return

class Implementation(Base):

    _value = 'Default value'

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, newvalue):
        self._value = newvalue

i = Implementation()
print 'Implementation.value:', i.value

i.value = 'New value'
print 'Changed value:', i.value

```

Base 和 Implementation 类中的方法都命名为 value(), 不过它们有不同的签名。

```
$ python abc_abstractproperty_rw_deco.py
```

```
Implementation.value: Default value
Changed value: New value
```

参见:

abc (<http://docs.python.org/library/abc.html>) 这个模块的标准库文档。

PEP 3119 (www.python.org/dev/peps/pep-3119) 介绍抽象基类。

collections (2.1 节) collections 模块包含很多集合类型的抽象基类。

PEP 3141 (www.python.org/dev/peps/pep-3141) 数字的一个类型层次结构。

Strategy pattern (http://en.wikipedia.org/wiki/Strategy_pattern) 策略模式的描述和示例, 这是一个常用的插件实现模式。

Plugins and monkeypatching (<http://us.pycon.org/2009/conference/schedule/event/47/>) Dr. André Roberge 提供的 PyCon 2009 演示文稿。

18.3 dis——Python 字节码反汇编工具

作用: 将代码对象转换为人类可读的字节码表示, 以便分析。

Python 版本: 1.4 及以后版本

`dis` 模块包括一些函数来处理 Python 字节码，可以将字节码“反汇编”为更可读的形式。查看解释器执行的字节码是一种手控调整紧密循环（tight loop）的好办法，还有助于完成其他优化。这个模块对于查找多线程应用中的竞态条件也很有用，因为可以用它来估计代码中哪一点可能切换线程控制。

警告：字节码的使用是 CPython 解释器的一个实现细节，特定于具体版本。针对你使用的解释器版本，参考其源代码中的 `Include/opcode.h`，查找字节码的正式名列表（canonical list）。

18.3.1 基本反汇编

函数 `dis()` 会输出一个 Python 代码源（模块、类、方法、函数或代码对象）的反汇编表示。可以从命令行运行 `dis` 对类似 `dis_simple.py` 的模块反汇编。

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  my_dict = { 'a':1 }
```

输出按列组织，包括原始源代码行号、代码对象中的指令“地址”、操作码名，以及传至操作码的所有参数。

```
$ python -m dis dis_simple.py

4          0 BUILD_MAP                1
          3 LOAD_CONST                0 (1)
          6 LOAD_CONST                1 ('a')
          9 STORE_MAP
         10 STORE_NAME                0 (my_dict)
         13 LOAD_CONST                2 (None)
         16 RETURN_VALUE
```

在这里，源代码转换为 5 个不同的操作来创建和填充字典，然后将结果保存到一个局部变量。由于 Python 解释器是基于栈的，因此前几步是用 `LOAD_CONST` 将常量按正确的顺序放入栈中，然后使用 `STORE_MAP` 弹出要添加到字典的新键和值。用 `STORE_NAME` 将所得到的对象绑定到名“`my_dict`”。

18.3.2 反汇编函数

遗憾的是，反汇编整个模块时不会自动递归反汇编函数。

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  def f(*args):
5      nargs = len(args)
6      print nargs, args
```

```

7
8 if __name__ == '__main__':
9     import dis
10    dis.dis(f)

```

反汇编 `dis_function.py` 的结果显示了将函数代码对象加载到栈然后转换为一个函数的操作 (LOAD_CONST, MAKE_FUNCTION), 不过没有函数体。

```
$ python -m dis dis_function.py
```

```

4          0 LOAD_CONST          0 (<code object f at 0x1
00479030, file "dis_function.py", line 4>)
          3 MAKE_FUNCTION          0
          6 STORE_NAME             0 (f)

8          9 LOAD_NAME           1 (__name__)
         12 LOAD_CONST             1 ('__main__')
         15 COMPARE_OP            2 (==)
         18 POP_JUMP_IF_FALSE     49

9          21 LOAD_CONST          2 (-1)
         24 LOAD_CONST            3 (None)
         27 IMPORT_NAME           2 (dis)
         30 STORE_NAME            2 (dis)

10         33 LOAD_NAME           2 (dis)
         36 LOAD_ATTR             2 (dis)
         39 LOAD_NAME            0 (f)
         42 CALL_FUNCTION         1
         45 POP_TOP
         46 JUMP_FORWARD           0 (to 49)
      >>  49 LOAD_CONST            3 (None)
         52 RETURN_VALUE

```

要查看函数内部, 必须把函数传递给 `dis()`。

```
$ python dis_function.py
```

```

5          0 LOAD_GLOBAL          0 (len)
          3 LOAD_FAST               0 (args)
          6 CALL_FUNCTION           1
          9 STORE_FAST              1 (nargs)

6         12 LOAD_FAST              1 (nargs)
         15 PRINT_ITEM
         16 LOAD_FAST              0 (args)
         19 PRINT_ITEM
         20 PRINT_NEWLINE
         21 LOAD_CONST             0 (None)

```



24 RETURN_VALUE

18.3.3 类

可以把类传递给 `dis()`，在这种情况下，会依次反汇编类的所有方法。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import dis
5
6  class MyObject(object):
7      """Example for dis."""
8
9      CLASS_ATTRIBUTE = 'some value'
10
11     def __str__(self):
12         return 'MyObject(%s)' % self.name
13
14     def __init__(self, name):
15         self.name = name
16
17 dis.dis(MyObject)

```

方法以字母顺序列出，而不是按它们在文件中出现的顺序。

```
$ python dis_class.py
```

Disassembly of `__init__`:

15	0	LOAD_FAST	1	(name)
	3	LOAD_FAST	0	(self)
	6	STORE_ATTR	0	(name)
	9	LOAD_CONST	0	(None)
	12	RETURN_VALUE		

Disassembly of `__str__`:

12	0	LOAD_CONST	1	('MyObject(%s)')
	3	LOAD_FAST	0	(self)
	6	LOAD_ATTR	0	(name)
	9	BINARY_MODULO		
	10	RETURN_VALUE		

18.3.4 使用反汇编进行调试

调试一个异常时，有时要查看是哪个字节码导致了问题，这可能很有用。要对一个错误周围的代码反汇编，有两种方法。第一种方法是在交互式解释器中使用 `dis()` 报告最后一个异常。如果未向 `dis()` 传入任何参数，它会查找一个异常，并显示导致这个异常的栈顶的反汇编结果。

```
$ python
```

```
Python 2.6.2 (r262:71600, Apr 16 2009, 09:17:39)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import dis
>>> j = 4
>>> i = i + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined
>>> dis.distb()
1 -->      0 LOAD_NAME              0 (i)
           3 LOAD_CONST            0 (4)
           6 BINARY_ADD
           7 STORE_NAME            0 (i)
          10 LOAD_CONST            1 (None)
          13 RETURN_VALUE

>>>
```

行号后面的 --> 指示了导致错误的操作码 (opcode)。由于没有定义 `i` 变量，因此无法将这个名称关联的值加载到栈中。

程序还可以将 `traceback` 直接传递到 `distb()`，来输出一个活动 `traceback` 的有关信息。在这个例子中，有一个 `DivideByZero` 异常；不过由于这个公式有两个除法，并不清楚哪一部分为 0。

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  i = 1
5  j = 0
6  k = 3
7
8  # ... many lines removed ...
9
10 try:
11     result = k * (i / j) + (i / k)
12 except:
13     import dis
14     import sys
15     exc_type, exc_value, exc_tb = sys.exc_info()
16     dis.distb(exc_tb)
```

在反汇编版本中，值加载到栈时很容易发现有问题的值。会用 --> 突出显示这个出错的操作，前一行将 `j` 的值压入栈。

```
$ python dis_traceback.py
```

```
4          0 LOAD_CONST            0 (1)
```

	3 STORE_NAME	0 (i)
5	6 LOAD_CONST	1 (0)
	9 STORE_NAME	1 (j)
6	12 LOAD_CONST	2 (3)
	15 STORE_NAME	2 (k)
10	18 SETUP_EXCEPT	26 (to 47)
11	21 LOAD_NAME	2 (k)
	24 LOAD_NAME	0 (i)
	27 LOAD_NAME	1 (j)
-->	30 BINARY_DIVIDE	
	31 BINARY_MULTIPLY	
	32 LOAD_NAME	0 (i)
	35 LOAD_NAME	2 (k)
	38 BINARY_DIVIDE	
	39 BINARY_ADD	
	40 STORE_NAME	3 (result)

...trimmed...

18.3.5 循环的性能分析

除了调试错误外，dis 还有助于发现性能问题。检查反汇编的代码对于紧密循环尤其有用，在这些循环中，Python 指令很少，但是它们会转换为一组效率很低的字节码。可以通过查看一个类 Dictionary 的不同实现来了解反汇编提供的帮助，这个类会读取一个单词列表，然后按其首字母分组。

```
import dis
import sys
import timeit

module_name = sys.argv[1]
module = __import__(module_name)
Dictionary = module.Dictionary

dis.dis(Dictionary.load_data)
print
t = timeit.Timer(
    'd = Dictionary(words)',
    """from %(module_name)s import Dictionary
words = [l.strip() for l in open('/usr/share/dict/words', 'rt')]
""" % locals()
)
```

```
iterations = 10
print 'TIME: %0.4f' % (t.timeit(iterations)/iterations)
```

可以用测试驱动应用 `dis_test_loop.py` 来运行 `Dictionary` 类的各个实现。

首先是 `Dictionary` 的一个简单（但很慢）的实现，如下所示。

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  class Dictionary(object):
5
6      def __init__(self, words):
7          self.by_letter = {}
8          self.load_data(words)
9
10     def load_data(self, words):
11         for word in words:
12             try:
13                 self.by_letter[word[0]].append(word)
14             except KeyError:
15                 self.by_letter[word[0]] = [word]
```

用这个版本运行测试程序时，会显示反汇编的程序，以及它运行所花费的时间。

```
$ python dis_test_loop.py dis_slow_loop
```

```
11          0 SETUP_LOOP                84 (to 87)
          3 LOAD_FAST                    1 (words)
          6 GET_ITER
      >>    7 FOR_ITER                      76 (to 86)
          10 STORE_FAST                  2 (word)

12          13 SETUP_EXCEPT           28 (to 44)

13          16 LOAD_FAST                  0 (self)
          19 LOAD_ATTR                    0 (by_letter)
          22 LOAD_FAST                    2 (word)
          25 LOAD_CONST                  1 (0)
          28 BINARY_SUBSCR
          29 BINARY_SUBSCR
          30 LOAD_ATTR                    1 (append)
          33 LOAD_FAST                    2 (word)
          36 CALL_FUNCTION                1
          39 POP_TOP
          40 POP_BLOCK
          41 JUMP_ABSOLUTE                7

14      >>    44 DUP_TOP
          45 LOAD_GLOBAL                  2 (KeyError)
```

```

48 COMPARE_OP          10 (exception match)
51 JUMP_IF_FALSE       27 (to 81)
54 POP_TOP
55 POP_TOP
56 POP_TOP
57 POP_TOP

15 58 LOAD_FAST          2 (word)
61 BUILD_LIST          1
64 LOAD_FAST           0 (self)
67 LOAD_ATTR           0 (by_letter)
70 LOAD_FAST           2 (word)
73 LOAD_CONST          1 (0)
76 BINARY_SUBSCR
77 STORE_SUBSCR
78 JUMP_ABSOLUTE       7
>> 81 POP_TOP
82 END_FINALLY
83 JUMP_ABSOLUTE       7
>> 86 POP_BLOCK
>> 87 LOAD_CONST        0 (None)
90 RETURN_VALUE

```

TIME: 0.1074

前面的输出显示 `dis_slow_loop.py` 花费了 0.107 4 秒来加载 OS X 上 `/usr/share/dict/words` 副本中的 234 936 个单词。这不算太坏，不过相应的反汇编结果显示出循环做了很多不必要的工作。它在操作码 13 处进入循环时，建立了一个异常上下文 (SETUP_EXCEPT)。然后在将 `word` 追加到列表之前，它使用了 6 个操作码来查找 `self.by_letter[word[0]]`。如果由于 `word[0]` 不在字典中而导致一个异常，异常处理程序会做完全相同的工作来确定 `word[0]` (3 个操作码)，并把 `self.by_letter[word[0]]` 设置为包含这个单词的一个新列表。

要避免建立这个异常，一种技术是对应字母表中的各个字母分别用一个列表预填充字典 `self.by_letter`。这意味着总会找到新单词相应的列表，可以在查找之后保存值。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import string
5
6  class Dictionary(object):
7
8      def __init__(self, words):
9          self.by_letter = dict( (letter, [])
10                                for letter in string.letters)
11          self.load_data(words)
12

```

```

13     def load_data(self, words):
14         for word in words:
15             self.by_letter[word[0]].append(word)

```

这个修改将操作码数减至一半，不过时间只减少到 0.098 4 秒。显然，异常处理有一些开销，但并不多。

```
$ python dis_test_loop.py dis_faster_loop
```

```

14          0 SETUP_LOOP                38 (to 41)
          3 LOAD_FAST                    1 (words)
          6 GET_ITER
    >>      7 FOR_ITER                    30 (to 40)
          10 STORE_FAST                   2 (word)

15          13 LOAD_FAST                  0 (self)
          16 LOAD_ATTR                    0 (by_letter)
          19 LOAD_FAST                    2 (word)
          22 LOAD_CONST                   1 (0)
          25 BINARY_SUBSCR
          26 BINARY_SUBSCR
          27 LOAD_ATTR                    1 (append)
          30 LOAD_FAST                    2 (word)
          33 CALL_FUNCTION                1
          36 POP_TOP
          37 JUMP_ABSOLUTE               7
    >>      40 POP_BLOCK
    >>      41 LOAD_CONST                  0 (None)
          44 RETURN_VALUE

```

```
TIME: 0.0984
```

将 `self.by_letter` 的查找移到循环之外（毕竟值没有改变），可以进一步提高性能。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import collections
5
6  class Dictionary(object):
7
8      def __init__(self, words):
9          self.by_letter = collections.defaultdict(list)
10         self.load_data(words)
11
12     def load_data(self, words):
13         by_letter = self.by_letter
14         for word in words:
15             by_letter[word[0]].append(word)

```



现在操作码 0-6 会查找 `self.by_letter` 的值，并把它另存为一个局部变量 `by_letter`。使用局部变量只需要一个操作码，而不是两个（语句 22 使用 `LOAD_FAST` 将字典放在栈中）。做了这个修改之后，运行时间降至 0.084 2 秒。

```
$ python dis_test_loop.py dis_fastest_loop
```

```

13          0 LOAD_FAST          0 (self)
            3 LOAD_ATTR          0 (by_letter)
            6 STORE_FAST         2 (by_letter)

14          9 SETUP_LOOP        35 (to 47)
            12 LOAD_FAST          1 (words)
            15 GET_ITER
    >>      16 FOR_ITER          27 (to 46)
            19 STORE_FAST         3 (word)

15          22 LOAD_FAST          2 (by_letter)
            25 LOAD_FAST          3 (word)
            28 LOAD_CONST         1 (0)
            31 BINARY_SUBSCR
            32 BINARY_SUBSCR
            33 LOAD_ATTR          1 (append)
            36 LOAD_FAST          3 (word)
            39 CALL_FUNCTION      1
            42 POP_TOP
            43 JUMP_ABSOLUTE      16
    >>      46 POP_BLOCK
    >>      47 LOAD_CONST          0 (None)
            50 RETURN_VALUE

```

```
TIME: 0.0842
```

Brandon Rhodes 还建议了进一步的优化，可以完全消除 Python 版本的 for 循环。如果使用 `itertools.groupby()` 来整理输入，将把迭代处理移至 C。这个方法很安全，因为输入已经是有序的。如果并非如此，程序则需要先进行排序。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import operator
5  import itertools
6
7  class Dictionary(object):
8
9      def __init__(self, words):
10         self.by_letter = {}
11         self.load_data(words)
12

```

```

13     def load_data(self, words):
14         # Arrange by letter
15         grouped = itertools.groupby(words, key=operator.itemgetter(0))
16         # Save arranged sets of words
17         self.by_letter = dict((group[0][0], group) for group in grouped)

```

这个 `itertools` 版本运行只需要 0.054 3 秒，仅为原时间的一半稍多。

```
$ python dis_test_loop.py dis_eliminate_loop
```

```

15          0 LOAD_GLOBAL              0 (itertools)
          3 LOAD_ATTR                    1 (groupby)
          6 LOAD_FAST                    1 (words)
          9 LOAD_CONST                  1 ('key')
         12 LOAD_GLOBAL              2 (operator)
         15 LOAD_ATTR                    3 (itemgetter)
         18 LOAD_CONST                  2 (0)
         21 CALL_FUNCTION              1
         24 CALL_FUNCTION            257
         27 STORE_FAST                2 (grouped)

17          30 LOAD_GLOBAL              4 (dict)
          33 LOAD_CONST                  3 (<code object
<genexpr> at 0x7e7b8, file "dis_eliminate_loop.py", line 17>)
          36 MAKE_FUNCTION              0
          39 LOAD_FAST                2 (grouped)
          42 GET_ITER
          43 CALL_FUNCTION              1
          46 CALL_FUNCTION              1
          49 LOAD_FAST                0 (self)
          52 STORE_ATTR                5 (by_letter)
          55 LOAD_CONST                0 (None)
          58 RETURN_VALUE

```

```
TIME: 0.0543
```

18.3.6 编译器优化

对编译的源代码反汇编还能发现一些编译器优化。例如，如果可能，字面量表达式会在编译时折叠。

```

1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  # Folded
5  i = 1 + 2
6  f = 3.4 * 5.6
7  s = 'Hello,' + ' World!'

```

```

8
9 # Not folded
10 I = i * 3 * 4
11 F = f / 2 / 3
12 S = s + '\n' + 'Fantastic!'

```

5~7 行表达式中的值不会改变完成操作的方式，所以可以在编译时计算表达式的结果，并压缩为单条 `LOAD_CONST` 指令。10~12 行则不同。因为这些表达式中涉及一个变量，而且这个变量指示的对象可能会覆盖所涉及的操作符，所以计算必须延迟到运行时。

```
$ python -m dis dis_constant_folding.py
```

```

5          0 LOAD_CONST          11 (3)
          3 STORE_NAME           0 (i)

6          6 LOAD_CONST          12 (19.04)
          9 STORE_NAME           1 (f)

7         12 LOAD_CONST          13 ('Hello, World!')
         15 STORE_NAME           2 (s)

10         18 LOAD_NAME           0 (i)
         21 LOAD_CONST           6 (3)
         24 BINARY_MULTIPLY
         25 LOAD_CONST           7 (4)
         28 BINARY_MULTIPLY
         29 STORE_NAME           3 (I)

11         32 LOAD_NAME           1 (f)
         35 LOAD_CONST           1 (2)
         38 BINARY_DIVIDE
         39 LOAD_CONST           6 (3)
         42 BINARY_DIVIDE
         43 STORE_NAME           4 (F)

12         46 LOAD_NAME           2 (s)
         49 LOAD_CONST           8 ('\n')
         52 BINARY_ADD
         53 LOAD_CONST           9 ('Fantastic!')
         56 BINARY_ADD
         57 STORE_NAME           5 (S)
         60 LOAD_CONST          10 (None)
         63 RETURN_VALUE

```

参见：

`dis` (<http://docs.python.org/library/dis.html>) 这个模块的标准库文档，包括字节码指令列表 (<http://docs.python.org/library/dis.html#python-bytecode-instructions>)。

Include/opcode.h CPython 解释器的源代码在 opcode.h 中定义了字节码。

Python Essential Reference, 第 4 版, David M. Beazley (www.informit.com/store/product.aspx?isbn=0672329786)

Python disassembly (<http://thomas.apestaart.org/log/?p=927>) 简单讨论了 Python 2.5 和 2.6 在字典中存储值的差别。

Why is looping over range() in Python faster than using a while loop? (<http://stackoverflow.com/questions/869229/why-is-looping-over-range-inpython-faster-than-using-a-while-loop>) StackOverflow.com 上对于通过反汇编字节码比较两个循环示例的讨论。

Decorator for binding constants at compile time (<http://code.activestate.com/recipes/277940/>) Raymond Hettinger 和 Skip Montanaro 提供的 Python 实用技巧, 包含一个函数装饰符, 它会重写一个函数的字节码, 插入全局常量来避免在运行时查找名称。

18.4 inspect——检查现场对象

作用: inspect 模块提供了一些函数, 用来对现场对象及其源代码完成自省。

Python 版本: 2.1 及以后版本

inspect 模块提供了一些函数来了解现场对象, 包括模块、类、实例、函数和方法。这个模块中的函数可以用来获取一个函数的原始源代码、查看栈中一个方法的参数, 以及抽取对生成源代码库文档有用的信息。

18.4.1 示例模块

本节余下的例子都会使用这个示例文件 example.py。

```
#!/usr/bin/env python

# This comment appears first
# and spans 2 lines.

# This comment does not show up in the output of getcomments().

"""Sample file to serve as the basis for inspect examples.
"""
def module_level_function(arg1, arg2='default', *args, **kwargs):
    """This function is declared in the module."""
    local_variable = arg1

class A(object):
    """The A class."""
    def __init__(self, name):
        self.name = name

    def get_name(self):
```

```

        "Returns the name of the instance."
        return self.name

instance_of_a = A('sample_instance')

class B(A):
    """This is the B class.
    It is derived from A.
    """

    # This method is not part of A.
    def do_something(self):
        """Does some work"""

    def get_name(self):
        "Overrides version from A"
        return 'B(' + self.name + ')'
```

18.4.2 模块信息

第一种自省是探查和了解现场对象。例如，可以发现模块中的类和函数、类的方法等。

要确定解释器如何作为模块处理和加载文件，可以使用 `getmoduleinfo()`。传入一个文件名作为唯一的参数，返回值是一个 tuple，其中包含模块库名、文件后缀、读取文件时使用的模式，以及 `imp` 模块中定义的模块类型。需要指出的重要的一点是，这个函数只查看文件的名称，而不会具体检查文件是否存在，也不会尝试读取这个文件。

```

import imp
import inspect
import sys
if len(sys.argv) >= 2:
    filename = sys.argv[1]
else:
    filename = 'example.py'

try:
    (name, suffix, mode, mtype) = inspect.getmoduleinfo(filename)
except TypeError:
    print 'Could not determine module type of %s' % filename
else:
    mtype_name = { imp.PY_SOURCE: 'source',
                   imp.PY_COMPILED: 'compiled',
                   }.get(mtype, mtype)

    mode_description = { 'rb': '(read-binary)',
                        'U': '(universal newline)',
                        }.get(mode, '')
```

```

print 'NAME      :', name
print 'SUFFIX    :', suffix
print 'MODE      :', mode, mode_description
print 'MTYPE     :', mtype_name

```

以下是运行一些示例的结果。

```
$ python inspect_getmoduleinfo.py example.py
```

```

NAME      : example
SUFFIX    : .py
MODE      : U (universal newline)
MTYPE     : source

```

```
$ python inspect_getmoduleinfo.py readme.txt
```

```
Could not determine module type of readme.txt
```

```
$ python inspect_getmoduleinfo.py notthere.pyc
```

```

NAME      : notthere
SUFFIX    : .pyc
MODE      : rb (read-binary)
MTYPE     : compiled

```

18.4.3 检查模块

还可以使用 `getmembers()` 检查现场对象，确定其组成部分。这个函数的参数是一个待扫描的对象（模块、类或实例）和一个谓词函数（可选），这个谓词函数用来过滤返回的对象。返回值是一个元组列表，元组包含两个值：成员名和成员的类型。`inspect` 模块包括很多这样的谓词函数（名为 `ismodule()`、`isclass()` 等）。

可能返回的成员类型取决于所扫描对象的类型。模块可能包含类和函数；类可能包含方法和属性，等等。

```

import inspect

import example

for name, data in inspect.getmembers(example):
    if name.startswith('__'):
        continue
    print '%s : %r' % (name, data)

```

这个例子会输出 `example` 模块的成员。模块有一些私有属性作为导入实现的一部分，另外还包含一组 `__builtins__`。在这个例子的输出中将忽略所有这些成员，因为它们不能真正算是模块的一部分，而且这个列表很长。

```
$ python inspect_getmembers_module.py
```

```
A : <class 'example.A'>
B : <class 'example.B'>
instance_of_a : <example.A object at 0x1004ddd10>
module_level_function : <function module_level_function at
0x1004cd050>
```

可以用谓词 (predicate) 参数过滤返回对象的类型。

```
import inspect
```

```
import example
```

```
for name, data in inspect.getmembers(example, inspect.isclass):
    print '%s : ' % name, repr(data)
```

现在输出中只包括类。

```
$ python inspect_getmembers_module_class.py
```

```
A : <class 'example.A'>
B : <class 'example.B'>
```

18.4.4 检查类

类似于模块，可以采用同样的方式使用 getmembers() 扫描类，不过成员的类型不同。

```
import inspect
from pprint import pprint
```

```
import example
```

```
pprint(inspect.getmembers(example.A), width=65)
```

由于没有应用过滤，因此输出显示了属性、方法、槽和类的其他成员。

```
$ python inspect_getmembers_class.py
```

```
[('__class__', <type 'type'>),
 ('__delattr__',
  <slot wrapper '__delattr__' of 'object' objects>),
 ('__dict__', <dictproxy object at 0x1004d0da8>),
 ('__doc__', 'The A class.'),
 ('__format__', <method '__format__' of 'object' objects>),
 ('__getattribute__',
  <slot wrapper '__getattribute__' of 'object' objects>),
 ('__hash__', <slot wrapper '__hash__' of 'object' objects>),
 ('__init__', <unbound method A.__init__>),
 ('__module__', 'example'),
 ('__new__',
  <built-in method __new__ of type object at 0x100187800>),
 ('__reduce__', <method '__reduce__' of 'object' objects>),
 ('__reduce_ex__',
```

```

    <method '__reduce_ex__' of 'object' objects>),
    ('__repr__', <slot wrapper '__repr__' of 'object' objects>),
    ('__setattr__',
     <slot wrapper '__setattr__' of 'object' objects>),
    ('__sizeof__', <method '__sizeof__' of 'object' objects>),
    ('__str__', <slot wrapper '__str__' of 'object' objects>),
    ('__subclasshook__',
     <built-in method __subclasshook__ of type object at 0x100385a10>),
    ('__weakref__', <attribute '__weakref__' of 'A' objects>),
    ('get_name', <unbound method A.get_name>)]

```

要查找一个类的成员，可以使用谓词 `ismethod()`。

```

import inspect
from pprint import pprint

import example

pprint(inspect.getmembers(example.A, inspect.ismethod))

```

现在只会返回未绑定的方法。

```
$ python inspect_getmembers_class_methods.py
```

```

[('__init__', <unbound method A.__init__>),
 ('get_name', <unbound method A.get_name>)]

```

B 的输出包括覆盖的 `get_name()` 方法、新方法，以及从 A 继承的 `__init__()` 方法。

```

import inspect
from pprint import pprint

import example

pprint(inspect.getmembers(example.B, inspect.ismethod))

```

从 A 继承的方法（如 `__init__()`）会标识为 B 的方法。

```
$ python inspect_getmembers_class_methods_b.py
```

```

[('__init__', <unbound method B.__init__>),
 ('do_something', <unbound method B.do_something>),
 ('get_name', <unbound method B.get_name>)]

```

18.4.5 文档串

可以用 `getdoc()` 获取一个对象的 docstring。返回值是 `__doc__` 属性，其中的制表符扩展为空格，并保证缩进一致。

```

import inspect
import example

```



```

print 'B.__doc__:'
print example.B.__doc__
print
print 'getdoc(B):'
print inspect.getdoc(example.B)

```

通过这个属性直接获取 docstring 的第二行时，它会缩进，不过 getdoc() 会把它移至左边界。

```
$ python inspect_getdoc.py
```

```

B.__doc__:
This is the B class.
    It is derived from A.

```

```

getdoc(B):
This is the B class.
It is derived from A.

```

除了具体的 docstring 外，还可以从实现对象的源文件获取注释（如果可以得到源文件）。

getcomments() 函数会查看对象的源文件，查找实现代码前面的注释。

```

import inspect
import example

print inspect.getcomments(example.B.do_something)

```

返回的行包括注释前缀，这里会去除空白符前缀。

```
$ python inspect_getcomments_method.py
```

```
# This method is not part of A.
```

将一个模块传入 getcomments() 时，返回值总是模块中的第一个注释。

```

import inspect
import example

print inspect.getcomments(example)

```

示例文件中邻接的行会作为一个注释，不过一旦出现一个空行，注释就会停止。

```
$ python inspect_getcomments_module.py
```

```

# This comment appears first
# and spans 2 lines.

```

18.4.6 获取源代码

如果可以得到一个模块的 .py 文件，可以使用 getsource() 和 getsourcelines() 获取类或方法的原始源代码。

```
import inspect
import example
```

```
print inspect.getsource(example.A)
```

传入一个类时，输出中会包含这个类的所有方法。

```
$ python inspect_getsource_class.py
```

```
class A(object):
    """The A class."""
    def __init__(self, name):
        self.name = name
    def get_name(self):
        "Returns the name of the instance."
        return self.name
```

要获取一个方法的源代码，可以将这个方法引用传入 `getsource()`。

```
import inspect
import example
```

```
print inspect.getsource(example.A.get_name)
```

在这里，会保留原来的缩进层次。

```
$ python inspect_getsource_method.py
```

```
def get_name(self):
    "Returns the name of the instance."
    return self.name
```

如果使用 `getsourcelines()` 而不是 `getsource()` 获取源代码行，会分解为单独的字符串。

```
import inspect
import pprint
import example
```

```
pprint.pprint(inspect.getsourcelines(example.A.get_name))
```

`getsourcelines()` 的返回值是一个 tuple，其中包含一个字符串列表（源文件中的代码行）和文件中源代码出现的起始行号。

```
$ python inspect_getsourcelines_method.py
```

```
(['    def get_name(self):\n',
  '        "Returns the name of the instance."\n',
  '        return self.name\n'],
 20)
```

如果不能得到源文件，`getsource()` 和 `getsourcelines()` 会产生一个 `IOError`。

18.4.7 方法和函数参数

除了函数或方法的文档外，还可以请求得到可调用方法的参数规范，包括默认值。`getargspec()` 函数会返回一个元组，其中包含位置参数名列表、可变位置参数名（例如 `*args`），可变命名参数名（例如，`**kwargs`）以及参数的默认值。如果有默认值，则对应位置参数列表的末尾参数。

```
import inspect
import example

arg_spec = inspect.getargspec(example.module_level_function)
print 'NAMES      :', arg_spec[0]
print '*'        :', arg_spec[1]
print '**       :', arg_spec[2]
print 'defaults:', arg_spec[3]

args_with_defaults = arg_spec[0][-len(arg_spec[3]):]
print 'args & defaults:', zip(args_with_defaults, arg_spec[3])
```

在下面这个例子中，函数的第一个参数 `arg1` 没有默认值。因此，惟一的默认值与 `arg2` 对应。

```
$ python inspect_getargspec_function.py
```

```
NAMES      : ['arg1', 'arg2']
*          : args
**         : kwargs
defaults: ('default',)
args & defaults: [('arg2', 'default')]
```

装饰符或其他函数可以用函数的参数规范来验证输入、提供不同的默认值等。不过，要编写一个合适的通用而且可重用的验证装饰符，存在一个特殊的挑战，因为对于同时接收命名参数和位置参数的函数来说，将接收到的参数与相应的参数名对应可能会很复杂。`getcallargs()` 提供了处理这种映射所需的逻辑。它会返回一个字典，其中填充了与指定函数参数名关联的参数。

```
import inspect
import example
import pprint
for args, kwds in [
    (('a',), {'unknown_name': 'value'}),
    (('a',), {'arg2': 'value'}),
    (('a', 'b', 'c', 'd'), {}),
    ((), {'arg1': 'a'}),
]:
    print args, kwds
    callargs = inspect.getcallargs(example.module_level_function,
                                   *args, **kwds)
    pprint.pprint(callargs, width=74)
    example.module_level_function(**callargs)
print
```

这个字典的键是函数的参数名，所以可以使用 ****** 语法来调用函数，将字典扩展到参数所在的栈。

```
$ python inspect_getcallargs.py

('a',) {'unknown_name': 'value'}
{'arg1': 'a',
 'arg2': 'default',
 'args': (),
 'kwargs': {'unknown_name': 'value'}}

('a',) {'arg2': 'value'}
{'arg1': 'a', 'arg2': 'value', 'args': (), 'kwargs': {}}

('a', 'b', 'c', 'd') {}
{'arg1': 'a', 'arg2': 'b', 'args': ('c', 'd'), 'kwargs': {}}

() {'arg1': 'a'}
{'arg1': 'a', 'arg2': 'default', 'args': (), 'kwargs': {}}
```

18.4.8 类层次结构

`inspect` 包含两个方法来直接处理类层次结构。第一个方法是 `getclasstree()`，它会基于给定的类及其基类创建一个类树的数据结构。在返回的列表中，各个元素可能是一个包含类及其基类的元组，也可能是另一个包含子类元组的列表。

```
import inspect
import example
class C(example.B):
    pass

class D(C, example.A):
    pass

def print_class_tree(tree, indent=-1):
    if isinstance(tree, list):
        for node in tree:
            print_class_tree(node, indent+1)
    else:
        print ' ' * indent, tree[0].__name__
    return

if __name__ == '__main__':
    print 'A, B, C, D:'
    print_class_tree(inspect.getclasstree([example.A, example.B, C, D]))
```

这个例子的输出是对应 A、B、C 和 D 类的继承“树”。D 出现了两次，因为它同时继承了 C 和 A。

```
$ python inspect_getclasstree.py
```

```
A, B, C, D:
object
  A
    D
    B
      C
        D
```

如果调用 `getclasstree()` 时将 `unique` 设置为值 `true`, 输出会有所不同。

```
import inspect
import example
from inspect_getclasstree import *

print_class_tree(inspect.getclasstree([example.A, example.B, C, D],
                                     unique=True,
                                     ))
```

这一次, D 在输出中只出现一次。

```
$ python inspect_getclasstree_unique.py
```

```
object
  A
    B
      C
        D
```

18.4.9 方法解析顺序

处理类层次结构的另一个函数是 `getmro()`, 它会返回一个类 tuple, 其中类的顺序为使用方法解析顺序 (Method Resolution Order, MRO) 解析一个可能从基类继承的属性时扫描各个类的顺序。在这个序列中, 每个类只出现一次。

```
import inspect
import example

class C(object):
    pass

class C_First(C, example.B):
    pass

class B_First(example.B, C):
    pass

print 'B_First:'
for c in inspect.getmro(B_First):
```



```

    print '\t', c.__name__
print
print 'C_First:'
for c in inspect.getmro(C_First):
    print '\t', c.__name__

```

这个输出展示了 MRO 搜索的“深度优先”特性。对于 B_First，按照搜索顺序 A 也在 C 之前，因为 B 派生自 A。

```
$ python inspect_getmro.py
```

```

B_First:
    B_First
    B
    A
    C
    object

C_First:
    C_First
    C
    B
    A
    object

```

18.4.10 栈与帧

除了代码对象的自省外，inspect 还包括一些函数以检查程序执行时的运行时环境。其中大多数函数都处理调用栈，操作对象为“调用帧”。栈中的每个帧记录是一个 6 元素的元组，包含帧对象、代码所在文件的文件名，该文件中当前运行的行的行号、所调用的函数名、源文件中上下文代码行的一个列表，以及该列表中当前行的索引。一般情况下，这些信息会在产生异常时用来建立 traceback。它对于记录日志或调试程序也很有用，因为可以查看栈帧来发现传入函数的参数值。

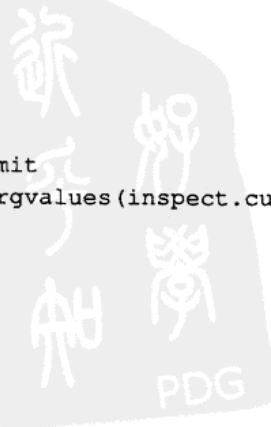
currentframe() 会返回位于栈顶的帧（对应当前函数）。getargvalues() 返回一个元组，包含参数名、变参名以及包含帧中局部值的一个字典。将它们结合起来，可以显示调用栈中不同点的函数参数和局部变量。

```

import inspect

def recurse(limit):
    local_variable = '.' * limit
    print limit, inspect.getargvalues(inspect.currentframe())
    if limit <= 0:
        return
    recurse(limit - 1)
    return

```



```
if __name__ == '__main__':
    recurse(2)
```

`local_variable` 的值包含在帧的局部变量中，尽管它并不是函数的一个参数。

```
$ python inspect_getargvalues.py
```

```
2 ArgInfo(args=['limit'], varargs=None, keywords=None,
locals={'local_variable': '.', 'limit': 2})
1 ArgInfo(args=['limit'], varargs=None, keywords=None,
locals={'local_variable': '.', 'limit': 1})
0 ArgInfo(args=['limit'], varargs=None, keywords=None,
locals={'local_variable': '', 'limit': 0})
```

使用 `stack()`，还可以访问从当前帧到第一个调用者的所有栈帧。这个例子与前面的例子类似，只不过它会一直等待，直到递归结束来输出栈信息。

```
import inspect
```

```
def show_stack():
    for level in inspect.stack():
        frame, filename, line_num, func, src_code, src_index = level
        print '%s[%d]\n -> %s' % (filename,
                                line_num,
                                src_code[src_index].strip(),
                                )
        print inspect.getargvalues(frame)
    print

def recurse(limit):
    local_variable = '.' * limit
    if limit <= 0:
        show_stack()
        return
    recurse(limit - 1)
    return
```

```
if __name__ == '__main__':
    recurse(2)
```

输出的最后一部分表示主程序，这在 `recurse()` 函数之外。

```
$ python inspect_stack.py
```

```
inspect_stack.py[9]
-> for level in inspect.stack():
ArgInfo(args=[], varargs=None, keywords=None,
locals={'src_index': 0, 'line_num': 9, 'frame': <frame object at
0x100360750>, 'level': (<frame object at 0x100360750>,
'inspect_stack.py', 9, 'show_stack', ['    for level in
```

```

inspect.stack():\n'], 0), 'src_code': ['    for level in
inspect.stack():\n'], 'filename': 'inspect_stack.py', 'func':
'show_stack'})

inspect_stack.py[21]
-> show_stack()
ArgInfo(args=['limit'], varargs=None, keywords=None,
locals={'local_variable': '', 'limit': 0})

inspect_stack.py[23]
-> recurse(limit - 1)
ArgInfo(args=['limit'], varargs=None, keywords=None,
locals={'local_variable': '.', 'limit': 1})

inspect_stack.py[23]
-> recurse(limit - 1)
ArgInfo(args=['limit'], varargs=None, keywords=None,
locals={'local_variable': '..', 'limit': 2})

inspect_stack.py[27]
-> recurse(2)
ArgInfo(args=[], varargs=None, keywords=None,
locals={'__builtins__': <module '__builtin__' (built-in)>,
'__file__': 'inspect_stack.py', 'inspect': <module 'inspect' from
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
inspect.pyc'>, 'recurse': <function recurse at 0x1004cd050>,
'__package__': None, '__name__': '__main__', 'show_stack':
<function show_stack at 0x1004def50>, '__doc__': 'Inspecting the
call stack.\n'})

```

还有一些函数可以构建不同上下文中的帧列表，如处理一个异常时。有关的更多细节参见 `trace()`、`getouterframes()` 和 `getinnerframes()` 的文档。

参见：

`inspect` (<http://docs.python.org/library/inspect.html>) 这个模块的标准库文档。

Python 2.3 Method Resolution Order (www.python.org/download/releases/2.3/mro/) Python 2.3 及以后版本使用的 C3 方法解析顺序的文档。

`pyclbr` (16.11 节) 通过解析模块而不是导入，`pyclbr` 模块也允许访问 `inspect` 同样的一些信息。

18.5 exceptions——内置异常类

作用：exceptions 模块定义了整个标准库以及解释器中使用的内置错误。

Python 版本：1.5 及以后版本

过去，Python 除了使用类作为异常外，还支持将简单的串消息作为异常。自 1.5 版本以来，所有标准库模块都使用类表示异常。从 Python 2.5 开始，串异常会导致一个 `DeprecationWarning`。

将来可能会完全去除对串异常的支持。

18.5.1 基类

异常类采用一个层次结构定义，见标准库文档中的描述。除了在组织方面明显的好处外，异常继承也很有用，因为相关的异常可以通过捕获其基类来捕获。大多数情况下，不会直接产生这些基类异常。

BaseException

所有异常的基类。实现了基本的逻辑，可以使用 `str()` 由传入构造函数的参数创建异常的一个串表示。

Exception

有些异常不会导致退出正在运行的应用，`Exception` 是所有这些异常的基类。用户定义的所有异常都应当使用 `Exception` 作为基类。

StandardError

标准库中使用的内置异常的基类。

ArithmeticError

与数学相关的错误的基类。

LookupError

无法找到某个对象时产生的错误的基类。

EnvironmentError

来自 Python 外部（操作系统、文件系统等）的错误的基类。

18.5.2 产生的异常

AssertionError

`AssertionError` 由一个失败的 `assert` 语句产生。

```
assert False, 'The assertion failed'
```

断言在库中很常见，用来强制对传入参数的约束。

```
$ python exceptions_AssertionError_assert.py
```

```
Traceback (most recent call last):
```

```
File "exceptions_AssertionError_assert.py", line 12, in <module>
```

```
    assert False, 'The assertion failed'
```

```
AssertionError: The assertion failed
```

通过类似 `failIf()` 等方法，`AssertionError` 还可以用在 `unittest` 模块创建的自动测试中。

```
import unittest
```

```
class AssertionExample(unittest.TestCase):
```

```
    def test(self):
        self.failUnless(False)
```

```
unittest.main()
```

运行自动测试套件的程序会监视 `AssertionError` 异常，作为测试失败的一个特殊提示。

```
$ python exceptions_AssertionError_unittest.py
```

```
F
```

```
=====
FAIL: test (__main__.AssertionExample)
-----
```

```
Traceback (most recent call last):
```

```
  File "exceptions_AssertionError_unittest.py", line 17, in test
    self.failUnless(False)
```

```
AssertionError: False is not True
```

```
-----
Ran 1 test in 0.000s
```

```
FAILED (failures=1)
```

AttributeError

当一个属性引用或赋值失败时，会产生 `AttributeError`。

```
class NoAttributes(object):
    pass
```

```
o = NoAttributes()
print o.attribute
```

这个例子展示了试图引用一个不存在的属性时会发生什么。

```
$ python exceptions_AttributeError.py
```

```
Traceback (most recent call last):
```

```
  File "exceptions_AttributeError.py", line 16, in <module>
    print o.attribute
```

```
AttributeError: 'NoAttributes' object has no attribute 'attribute'
```

大多数 Python 类都接受任意属性。类可以使用 `__slots__` 定义一组固定的属性，以节省内存，并提高性能。

```
class MyClass(object):
    __slots__ = ( 'attribute', )
```

```
o = MyClass()
o.attribute = 'known attribute'
o.not_a_slot = 'new attribute'
```

对一个定义了 `__slots__` 的类设置未知的属性，会导致一个 `AttributeError`。

```
$ python exceptions_AttributeError_slot.py
```

```
Traceback (most recent call last):
  File "exceptions_AttributeError_slot.py", line 15, in <module>
    o.not_a_slot = 'new attribute'
AttributeError: 'MyClass' object has no attribute 'not_a_slot'
```

程序试图修改一个只读属性时也会产生 `AttributeError`。

```
class MyClass(object):

    @property
    def attribute(self):
        return 'This is the attribute value'

o = MyClass()
print o.attribute
o.attribute = 'New value'
```

可以使用 `@property` 修饰符但不提供设置器函数来创建只读属性。

```
$ python exceptions_AttributeError_assignment.py
```

```
This is the attribute value
Traceback (most recent call last):
  File "exceptions_AttributeError_assignment.py", line 20, in
<module>
    o.attribute = 'New value'
AttributeError: can't set attribute
```

EOFError

对于类似 `input()` 或 `raw_input()` 这样的内置函数，如果在遇到输入流末尾之前没有读到任何数据，则会产生一个 `EOFError`。

```
while True:
    data = raw_input('prompt:')
    print 'READ:', data
```

文件方法 `read()` 遇到文件末尾时并不是产生一个异常，而会返回一个空串。

```
$ echo hello | python exceptions_EOFError.py
```

```
prompt:READ: hello
prompt:Traceback (most recent call last):
  File "exceptions_EOFError.py", line 13, in <module>
    data = raw_input('prompt:')
EOFError: EOF when reading a line
```

FloatingPointError

这个错误由导致错误的浮点操作产生，前提是已经打开了浮点异常控制 (fpectl)。启用 fpectl 时，要求编译解释器时提供 `--with-fpectl` 标志。不过，在标准库文档中并不鼓励使用 fpectl。

```
import math
import fpectl

print 'Control off:', math.exp(1000)
fpectl.turnon_sigfpe()
print 'Control on:', math.exp(1000)
```

GeneratorExit

调用一个生成器的 `close()` 方法时，会在其中产生一个 `GeneratorExit`。

```
def my_generator():
    try:
        for i in range(5):
            print 'Yielding', i
            yield i
    except GeneratorExit:
        print 'Exiting early'

g = my_generator()
print g.next()
g.close()
```

生成器应当捕获 `GeneratorExit`，在其提前终止时把它作为一个信号来完成清理。

```
$ python exceptions_GeneratorExit.py
```

```
Yielding 0
0
Exiting early
```

IOError

输入或输出失败时会产生这个错误，例如，如果磁盘已满，或者一个输入文件不存在。

```
try:
    f = open('/does/not/exist', 'r')
except IOError as err:
    print 'Formatted      :', str(err)
    print 'Filename      :', err.filename
    print 'Errno         :', err.errno
    print 'String error:', err.strerror
```

`filename` 属性包含出现错误的文件的名字。`errno` 属性是系统错误号，由平台的 C 库定义。对应 `errno` 的错误消息串保存在 `strerror` 中。

```
$ python exceptions_IOError.py
```

```
Formatted   : [Errno 2] No such file or directory: '/does/not/exist'
Filename    : /does/not/exist
Errno       : 2
String error: No such file or directory
```

ImportError

无法导入一个模块或模块中的一个成员时会产生这个异常。有些条件下会产生 ImportError。

```
import module_does_not_exist
```

如果一个模块不存在，导入系统会产生 ImportError。

```
$ python exceptions_ImportError_nomodule.py
```

```
Traceback (most recent call last):
  File "exceptions_ImportError_nomodule.py", line 12, in <module>
    import module_does_not_exist
ImportError: No module named module_does_not_exist
```

如果使用了 `from X import Y`，而在模块 X 中无法找到 Y，就会产生一个 ImportError。

```
from exceptions import MadeUpName
```

错误消息只包括缺少的名字，而没有显示从哪个模块或包加载它。

```
$ python exceptions_ImportError_missingname.py
```

```
Traceback (most recent call last):
  File "exceptions_ImportError_missingname.py", line 12, in
<module>
    from exceptions import MadeUpName
ImportError: cannot import name MadeUpName
```

IndexError

如果一个序列引用越界，就会产生 IndexError。

```
my_seq = [ 0, 1, 2 ]
print my_seq[3]
```

超出列表末尾的引用会导致一个错误。

```
$ python exceptions_IndexError.py
```

```
Traceback (most recent call last):
  File "exceptions_IndexError.py", line 13, in <module>
    print my_seq[3]
IndexError: list index out of range
```



KeyError

类似地，如果没有找到一个值作为字典的键，就会产生一个 `KeyError`。

```
d = { 'a':1, 'b':2 }
print d['c']
```

这条错误消息的文本就是所查找的键。

```
$ python exceptions_KeyError.py
```

```
Traceback (most recent call last):
  File "exceptions_KeyError.py", line 13, in <module>
    print d['c']
KeyError: 'c'
```

KeyboardInterrupt

用户按下 `Ctrl-C`（或 `Delete`）键终止一个正在运行的程序时，会出现一个 `KeyboardInterrupt`。不同于大多数其他异常，`KeyboardInterrupt` 直接继承自 `BaseException`，以避免被捕获 `Exception` 的全局异常处理程序所捕获。

```
try:
    print 'Press Return or Ctrl-C:',
    ignored = raw_input()
except Exception, err:
    print 'Caught exception:', err
except KeyboardInterrupt, err:
    print 'Caught KeyboardInterrupt'
else:
    print 'No exception'
```

在提示窗口按下 `Ctrl-C` 键会导致一个 `KeyboardInterrupt` 异常。

```
$ python exceptions_KeyboardInterrupt.py
```

```
Press Return or Ctrl-C: ^CCaught KeyboardInterrupt
```

MemoryError

如果一个程序用尽了所有内存，而且可以恢复（例如，通过删除一些对象），就会产生一个 `MemoryError`。

```
import itertools

# Try to create a MemoryError by allocating a lot of memory
l = []
for i in range(3):
    try:
        for j in itertools.count(1):
            print i, j
            l.append('' * (2**30))
```

```

except MemoryError:
    print '(error, discarding existing list)'
    l = []

```

当一个程序耗尽内存时，出错之后的行为可能不可预测。能不能构造一条错误消息就可能很成问题，因为创建这个字符串缓冲区也需要分配新的内存。

```
$ python exceptions_MemoryError.py
```

```

python(49670) malloc: *** mmap(size=1073745920) failed
(error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
python(49670) malloc: *** mmap(size=1073745920) failed
(error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
python(49670) malloc: *** mmap(size=1073745920) failed
(error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
0 1
0 2
0 3
(error, discarding existing list)
1 1
1 2
1 3
(error, discarding existing list)
2 1
2 2
2 3
(error, discarding existing list)

```

NameError

如果代码引用了一个名字，而当前作用域中不存在这个名字，就会产生 NameError 异常。未限定变量名就是这样一个例子。

```

def func():
    print unknown_name

```

```
func()
```

错误消息指出“全局名字”(global name)，这是因为查找名字时会从局部作用域开始，再向上在全局作用域中搜索，直至失败。

```
$ python exceptions_NameError.py
```

```
Traceback (most recent call last):
```

```

File "exceptions_NameError.py", line 15, in <module>
    func()
File "exceptions_NameError.py", line 13, in func
    print unknown_name
NameError: global name 'unknown_name' is not defined

```

NotImplementedError

用户自定义的基类可能产生 `NotImplementedError`，来指示一个方法或行为需要由子类定义，这是在模拟接口。

```

class BaseClass(object):
    """Defines the interface"""
    def __init__(self):
        super(BaseClass, self).__init__()
    def do_something(self):
        """The interface, not implemented"""
        raise NotImplementedError(
            self.__class__.__name__ + '.do_something'
        )
class SubClass(BaseClass):
    """Implementes the interface"""
    def do_something(self):
        """really does something"""
        print self.__class__.__name__ + ' doing something!'

```

```

SubClass().do_something()
BaseClass().do_something()

```

强制实现接口的另一种方法是使用 `abc` 模块创建一个抽象基类。

```
$ python exceptions_NotImplementedError.py
```

```

SubClass doing something!
Traceback (most recent call last):
  File "exceptions_NotImplementedError.py", line 29, in <module>
    BaseClass().do_something()
  File "exceptions_NotImplementedError.py", line 19, in do_something
    self.__class__.__name__ + '.do_something'
NotImplementedError: BaseClass.do_something

```

OSError

一个操作系统级函数返回错误时会产生 `OSError`。这是 `os` 模块中使用的主要错误类，`subprocess` 和其他提供操作系统接口的模块也会使用这个错误类。

```

import os

for i in range(10):
    try:

```



```

    print i, os.ttyname(i)
except OSError as err:
    print
    print '  Formatted      :', str(err)
    print '  Errno         :', err.errno
    print '  String error:', err.strerror
    break

```

对于 IOError、errno 和 strerror 属性会填入系统特定的值。filename 属性设置为 None。

```
$ python exceptions_OSError.py
```

```

0 /dev/tty0
1
Formatted      : [Errno 25] Inappropriate ioctl for device
Errno         : 25
String error: Inappropriate ioctl for device

```

OverflowError

当一个算术运算超出变量类型的界限时，会产生一个 OverflowError。长整数会随着值的增大分配更多内存，所以最后产生的错误是 MemoryError。常规整数将根据需要转换为长整数值。

```

import sys

print 'Regular integer: (maxint=%s)' % sys.maxint
try:
    i = sys.maxint * 3
    print 'No overflow for ', type(i), 'i =', i
except OverflowError, err:
    print 'Overflowed at ', i, err

print
print 'Long integer:'
for i in range(0, 100, 10):
    print '%2d' % i, 2L ** i

print
print 'Floating point values:'
try:
    f = 2.0**i
    for i in range(100):
        print i, f
        f = f ** 2
except OverflowError, err:
    print 'Overflowed after ', f, err

```

如果由乘法得到的一个整数不再适合作为常规整数，它会转换为一个长整数对象。在这个例子中，当值不能再用一个双精度浮点数表示时，使用浮点值的指数公式会溢出。

```
$ python exceptions_OverflowError.py

Regular integer: (maxint=9223372036854775807)
No overflow for <type 'long'> i = 27670116110564327421

Long integer:
0 1
10 1024
20 1048576
30 1073741824
40 1099511627776
50 1125899906842624
60 1152921504606846976
70 1180591620717411303424
80 1208925819614629174706176
90 1237940039285380274899124224

Floating-point values:
0 1.23794003929e+27
1 1.53249554087e+54
2 2.34854258277e+108
3 5.5156522631e+216
Overflowed after 5.5156522631e+216 (34, 'Result too large')
```

ReferenceError

使用一个 weakref 代理访问已经被垃圾回收的对象时，会出现一个 ReferenceError。

```
import gc
import weakref

class ExpensiveObject(object):
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print '(Deleting %s)' % self

obj = ExpensiveObject('obj')
p = weakref.proxy(obj)

print 'BEFORE:', p.name
obj = None
print 'AFTER:', p.name
```

这个例子会删除指向原对象 obj 的唯一的强引用，从而将这个对象删除。

```
$ python exceptions_ReferenceError.py
```

```
BEFORE: obj
```



```
(Deleting <__main__.ExpensiveObject object at 0x1004667d0>)
AFTER:
Traceback (most recent call last):
  File "exceptions_ReferenceError.py", line 26, in <module>
    print 'AFTER:', p.name
ReferenceError: weakly-referenced object no longer exists
```

RuntimeError

如果没有其他更特定的异常可用，就要使用 `RuntimeError` 异常。解释器自己不经常产生这个异常，但是一些用户代码会经常使用这个异常。

StopIteration

当一个迭代器完成工作时，它的 `next()` 方法会产生 `StopIteration`。一般不认为这个异常是一个错误。

```
l=[0,1,2]
i=iter(l)

print i
print i.next()
print i.next()
print i.next()
print i.next()
```

正常的 `for` 循环会捕获 `StopIteration` 异常，并跳出循环。

```
$ python exceptions_StopIteration.py

<listiterator object at 0x100459850>
0
1
2
Traceback (most recent call last):
  File "exceptions_StopIteration.py", line 19, in <module>
    print i.next()
StopIteration
```

SyntaxError

当解析器发现它无法理解的源代码时，会出现一个 `SyntaxError`。导入一个模块、调用 `exec` 或调用 `eval()` 时可能发生这种错误。

```
try:
    print eval('five times three')
except SyntaxError, err:
    print 'Syntax error %s (%s-%s): %s' % \
        (err.filename, err.lineno, err.offset, err.text)
    print err
```

可以使用异常属性来准确地查找输入文本中哪一部分导致了异常。

```
$ python exceptions_SyntaxError.py
```

```
Syntax error <string> (1-10): five times three
invalid syntax (<string>, line 1)
```

SystemError

如果错误发生在解释器本身，而且有机会继续成功运行，会产生一个 `SystemError`。系统错误通常指示解释器中存在一个 bug，应当向维护人员报告。

SystemExit

当调用 `sys.exit()` 时，它会产生 `SystemExit` 而不是立即退出。这就使得 `try:finally` 块中的清理代码得以运行，而且允许一些特殊环境（如调试工具和测试框架）捕获异常而避免退出。

TypeError

结合对象或者在对象上调用函数时，如果对象的类型不正确，会产生一个 `TypeError`。

```
result = 5 + 'string'
```

`TypeError` 和 `ValueError` 异常往往会混淆。`ValueError` 通常表示一个值的类型是正确的，但是越界了。而 `TypeError` 表示使用的对象类型不正确（也就是说，使用了一个整数而不是一个字符串）。

```
$ python exceptions_TypeError.py
```

```
Traceback (most recent call last):
  File "exceptions_TypeError.py", line 12, in <module>
    result = 5 + 'string'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

UnboundLocalError

`UnboundLocalError` 是一种 `NameError`，特别针对局部变量名。

```
def throws_global_name_error():
    print unknown_global_name

def throws_unbound_local():
    local_val = local_val + 1
    print local_val

try:
    throws_global_name_error()
except NameError, err:
    print 'Global name error:', err

try:
    throws_unbound_local()
```



```
except UnboundLocalError, err:
    print 'Local name error:', err
```

全局 NameError 和 UnboundLocal 之间的差别在于使用名字的方式不同。由于名字 “local_val” 出现在表达式的左边，因此它被解释为一个局部变量名。

```
$ python exceptions_UnboundLocalError.py
```

```
Global name error: global name 'unknown_global_name' is not
defined
Local name error: local variable 'local_val' referenced before
assignment
```

UnicodeError

UnicodeError 是 ValueError 的一个子类，出现 Unicode 问题时会产生这个错误。对应具体情况分别有 UnicodeEncodeError、UnicodeDecodeError 和 UnicodeTranslateError 子类。

ValueError

如果一个函数接收到的值类型正确，但是值不合法，就会使用 ValueError。

```
print chr(1024)
```

ValueError 异常是一个通用错误，在大量第三方库中都使用这个异常来指示向函数传入了一个非法的参数。

```
$ python exceptions_ValueError.py
```

```
Traceback (most recent call last):
  File "exceptions_ValueError.py", line 12, in <module>
    print chr(1024)
ValueError: chr() arg not in range(256)
```

ZeroDivisionError

当 0 用作除法运算的分母时，会产生一个 ZeroDivisionError。

```
print 'Division:',
try:
    print 1 / 0
except ZeroDivisionError as err:
    print err

print 'Modulo :',
try:
    print 1 % 0
except ZeroDivisionError as err:
    print err
```

当分母为 0 时，求余操作符也会产生 ZeroDivisionError。

```
$ python exceptions_ZeroDivisionError.py
```



Division: integer division or modulo by zero
Modulo : integer division or modulo by zero

18.5.3 警告类型

`exceptions` 模块还定义了一些用于 `warnings` 模块的异常。

`Warning` 所有警告的基类。

`UserWarning` 来自用户代码的警告的基类。

`DeprecationWarning` 用于不再维护的特性。

`PendingDeprecationWarning` 用于很快会废弃的特性。

`SyntaxWarning` 用于有问题的语法。

`RuntimeWarning` 用于运行时可能导致问题的事件。

`FutureWarning` 关于将来语言或库中可能的改变的警告。

`ImportWarning` 关于导入模块时出现的警告。

`UnicodeWarning` 关于 Unicode 文本中的警告。

参见：

`exceptions` (<http://docs.python.org/library/exceptions.html>) 这个模块的标准库文档。

`warnings` (18.1 节) 非错误警告消息。

`__slots__` 使用 `__slots__` 减少内存消耗的 Python 语言参考文档。

`abc` (18.2 节) 抽象基类。

`math` (5.4 节) `math` 模块有一些特殊的函数可以安全地完成浮点计算。

`weakref` (2.7 节) `weakref` 模块允许程序保留对象的引用而不会阻止垃圾回收。



第 19 章

模块与包

Python 的主要扩展机制使用了保存到模块的源代码，并通过 `import` 语句组合到程序中。大多数开发人员所认为的“Python”特性实际上是作为模块集合实现的，称为标准库，这也是本书讨论的主题。尽管导入特性是解释器本身内置的，不过库中还有很多与导入过程有关的模块。

`imp` 模块提供了解释器使用的导入机制的底层实现。它可以用来在运行时动态地导入模块，而不是使用 `import` 语句在启动时加载。如果不能提前知道需要导入的模块的名称，动态地加载模块就很有用，比如一个程序的插件或扩展包。

`zipimport` 为保存到 ZIP 归档中的模块和包提供了一个定制导入工具。例如，它可以用来加载 Python EGG 文件，还可以作为一种便利方法用来打包和发布应用。

Python 包除了包括源代码，还可以包括支持资源文件，如模板、默认配置文件、图像和其他数据。`pkgutil` 模块中实现了一个以可移植方式访问资源文件的接口。它还支持修改一个包的导入路径，从而可以将内容安装到多个目录中但是作为同一个包的不同部分出现。

19.1 `imp`——Python 的导入机制

作用：`imp` 模块提供了 Python `import` 语句的实现。

Python 版本：2.2.1 及以后版本

`imp` 模块包括一些函数，可以提供 Python 导入机制的部分底层实现，来加载包和模块中的代码。可以利用它动态导入模块，另外有些情况下，需要导入的模块名在编写代码时是未知的（例如，应用的插件或扩展包），这些情况下这个包也很有用。

19.1.1 示例包

本节中的例子使用了一个名为 `example` 的包，其中包含 `__init__.py`。

```
print 'Importing example package'
```

这些例子还使用了一个名为 `submodule` 的模块，其中包含以下内容：

```
print 'Importing submodule'
```

导入这个包或模块时，可以查看示例输出中 `print` 语句的文本。

19.1.2 模块类型

Python 支持多种类型的模块。打开模块并添加到命名空间时，每种类型的模块都需要其自

己的处理，另外不同平台对不同类型的支持也有所不同。例如，在 Microsoft Windows 下，共享库从扩展名为 .dll 或 .pyd 的文件加载，而不是 .so。使用解释器的调试（debug）构建版本而不是正常的发行（release）构建版本时，C 模块的扩展包也可能改变，因为这些扩展包编译时可能还包含调试信息。如果一个 C 扩展库或其他模块不能按预期加载，可以使用 `get_suffixes()` 输出当前平台支持的类型列表以及相应的加载参数。

```
import imp

module_types = { imp.PY_SOURCE: 'source',
                  imp.PY_COMPILED: 'compiled',
                  imp.C_EXTENSION: 'extension',
                  imp.PY_RESOURCE: 'resource',
                  imp.PKG_DIRECTORY: 'package',
                  }

def main():
    fmt = '%10s %10s %10s'
    print fmt % ('Extension', 'Mode', 'Type')
    print '-' * 32
    for extension, mode, module_type in imp.get_suffixes():
        print fmt % (extension, mode, module_types[module_type])

if __name__ == '__main__':
    main()
```

返回值是一个元组序列，其中包含文件扩展名、打开文件（包含有模块）所用的模式，以及一个类型码（模块中定义的一个常量）。这个表并不完备，因为有些可导入的模块或包类型并不对应于单个文件。

```
$ python imp_get_suffixes.py
```

Extension	Mode	Type
.so	rb	extension
module.so	rb	extension
.py	U	source
.pyc	rb	compiled

19.1.3 查找模块

加载模块的第一步是找到模块。`find_module()` 会扫描导入搜索路径，查找有指定名字的包或模块。它会返回一个打开的文件句柄（如果对这个类型适用）、包含这个模块的文件的名字，以及一个“描述”（这是一个元组，如 `get_suffixes()` 返回的元组）。

```
import imp
from imp_get_suffixes import module_types
import os

# Get the full name of the directory containing this module
```



```

base_dir = os.path.dirname(__file__) or os.getcwd()

print 'Package:'
f, pkg_fname, description = imp.find_module('example')
print module_types[description[2]], pkg_fname.replace(base_dir, '.')
print

print 'Submodule:'
f, mod_fname, description = imp.find_module('submodule', [pkg_fname])
print module_types[description[2]], mod_fname.replace(base_dir, '.')
if f: f.close()

```

`find_module()` 不处理加点的名字 (如 `example.submodule`), 所以调用者必须注意, 要为嵌套的模块传入正确的路径。这说明, 从包导入嵌套的模块时, 要提供一个指向包目录的路径, 使 `find_module()` 能够找到包中的一个模块。

```
$ python imp_find_module.py
```

```

Package:
package ./example

Submodule:
source ./example/submodule.py

```

如果 `find_module()` 无法找到这个模块, 它会产生一个 `ImportError`。

```

import imp

try:
    imp.find_module('no_such_module')
except ImportError, err:
    print 'ImportError:', err

```

错误消息会包括所缺少的模块的名字。

```
$ python imp_find_module_error.py
```

```
ImportError: No module named no_such_module
```

19.1.4 加载模块

找到模块后, 使用 `load_module()` 具体导入这个模块。`load_module()` 的参数包括一个完整的模块名 (加点的路径), 以及 `find_module()` 返回的值 (打开的文件句柄、文件名和描述元组)。

```

import imp

f, filename, description = imp.find_module('example')
try:
    example_package = imp.load_module('example', f,
                                       filename, description)
    print 'Package:', example_package

```

```
finally:
    if f:
        f.close()

f, filename, description = imp.find_module(
    'submodule', example_package.__path__)
try:
    submodule = imp.load_module('example.submodule', f,
                                filename, description)
    print 'Submodule:', submodule
finally:
    if f:
        f.close()
```

`load_module()` 用指定的名字创建一个新的模块对象，为它加载代码，并添加到 `sys.modules`。

```
$ python imp_load_module.py
```

```
Importing example package
Package: <module 'example' from '/Users/dhellmann/Documents/
PyMOTW/book/PyMOTW/imp/example/__init__.pyc'>
Importing submodule
Submodule: <module 'example.submodule' from '/Users/dhellmann/
Documents/PyMOTW/book/PyMOTW/imp/example/submodule.pyc'>
```

如果对一个已经导入的模块调用 `load_module()`，其效果就像是在现有的模块对象上调用 `reload()`。

```
import imp
import sys

for i in range(2):
    print i,
    try:
        m = sys.modules['example']
    except KeyError:
        print '(not in sys.modules)',
    else:
        print '(have in sys.modules)',
    f, filename, description = imp.find_module('example')
    example_package = imp.load_module('example', f, filename,
                                      description)
```

这种情况下不会创建一个新模块，而会替换现有模块的内容。

```
$ python imp_load_module_reload.py
```

```
0 (not in sys.modules) Importing example package
1 (have in sys.modules) Importing example package
```

参见：

`imp` (<http://docs.python.org/library/imp.html>) 这个模块的标准库文档。

17.2.6 节 sys 模块中的导入 hook、模块搜索路径以及其他相关的机制。

inspect (18.4 节) 通过编程从一个模块加载信息。

PEP 302 (www.python.org/dev/peps/pep-0302) 新的导入 hook。

PEP 369 (www.python.org/dev/peps/pep-0369) 将来的导入 hook。

19.2 zipimport——从 ZIP 归档加载 Python 代码

作用：导入作为 ZIP 归档成员保存的 Python 模块。

Python 版本：2.3 及以后版本

zipimport 模块实现了 zipimporter 类，这个类可以用来查找和加载 ZIP 归档中的 Python 模块。zipimporter 支持 PEP 302 中指定的“导入 hook”API；Python Eggs 就采用这种方式。

通常没有必要直接使用 zipimport 模块，因为只要归档出现在 sys.path 中，就可以直接从 ZIP 归档导入。不过，研究如何使用导入工具 API 对于学习可用的特性以及了解如何完成模块导入很有意义。另外如果用 zipfile.PyZipFile 创建作为 ZIP 归档打包的应用，了解 ZIP 导入工具如何工作还有助于调试发布这些应用时可能出现的问题。

19.2.1 示例

这些例子重用了讨论 zipfile 时的一些代码来创建一个示例 ZIP 归档，其中包含一些 Python 模块。

```
import sys
import zipfile

if __name__ == '__main__':
    zf = zipfile.PyZipFile('zipimport_example.zip', mode='w')
    try:
        zf.writepy('.')
        zf.write('zipimport_get_source.py')
        zf.write('example_package/README.txt')
    finally:
        zf.close()
    for name in zf.namelist():
        print name
```

运行其余例子之前先运行 zipimport_make_example.py，创建一个包含示例目录中所有模块的 ZIP 归档，以及本节中例子所需的一些测试数据。

```
$ python zipimport_make_example.py
```

```
__init__.pyc
example_package/__init__.pyc
zipimport_find_module.pyc
zipimport_get_code.pyc
zipimport_get_data.pyc
zipimport_get_data_nozip.pyc
```

```

zipimport_get_data_zip.pyc
zipimport_get_source.pyc
zipimport_is_package.pyc
zipimport_load_module.pyc
zipimport_make_example.pyc
zipimport_get_source.py
example_package/README.txt

```

19.2.2 查找模块

给定模块的全名，`find_module()` 会尝试在 ZIP 归档中查找这个模块。

```
import zipimport
```

```

importer = zipimport.zipimporter('zipimport_example.zip')

for module_name in [ 'zipimport_find_module', 'not_there' ]:
    print module_name, ':', importer.find_module(module_name)

```

如果找到了这个模块，会返回 `zipimporter` 实例。否则，返回 `None`。

```
$ python zipimport_find_module.py
```

```

zipimport_find_module : <zipimporter object "zipimport_example.zip">
not_there : None

```

19.2.3 访问代码

`get_code()` 方法会从归档中加载一个模块的代码对象。

```
import zipimport
```

```

importer = zipimport.zipimporter('zipimport_example.zip')
code = importer.get_code('zipimport_get_code')
print code

```

代码对象与 `module` 对象不同，不过可以用代码对象创建一个 `module` 对象。

```
$ python zipimport_get_code.py
```

```

<code object <module> at 0x1002cb130, file
"./zipimport_get_code.py", line 7>

```

要加载这个代码作为一个可用的模块，可以使用 `load_module()`。

```
import zipimport
```

```

importer = zipimport.zipimporter('zipimport_example.zip')
module = importer.load_module('zipimport_get_code')
print 'Name      :', module.__name__
print 'Loader   :', module.__loader__
print 'Code     :', module.code

```

结果会得到一个已配置的模块对象，就好像是从常规的 `import` 语句加载代码一样。

```
$ python zipimport_load_module.py

<code object <module> at 0x100431d30, file
"./zipimport_get_code.py", line 7>
Name      : zipimport_get_code
Loader    : <zipimporter object "zipimport_example.zip">
Code      : <code object <module> at 0x100431d30, file
"./zipimport_get_code.py", line 7>
```

19.2.4 源代码

类似于 inspect 模块，可以从 ZIP 归档获取一个模块的源代码（如果归档包含这个源代码）。对于这个例子，zipimport_example.zip 中只增加了 zipimport_get_source.py（其余模块都作为 .pyc 文件增加）。

```
import zipimport

importer = zipimport.zipimporter('zipimport_example.zip')
for module_name in ['zipimport_get_code', 'zipimport_get_source']:
    source = importer.get_source(module_name)
    print '=' * 80
    print module_name
    print '=' * 80
    print source
    print
```

如果不能得到一个模块的源代码，get_source() 会返回 None。

```
$ python zipimport_get_source.py

=====
zipimport_get_code
=====
None

=====
zipimport_get_source
=====
#!/usr/bin/env python
#
# Copyright 2007 Doug Hellmann.
#
"""Retrieving the source code for a module within a zip archive.
"""
#end_pymotw_header

import zipimport
```

```
importer = zipimport.zipimporter('zipimport_example.zip')
for module_name in ['zipimport_get_code', 'zipimport_get_source']:
    source = importer.get_source(module_name)
    print '=' * 80
    print module_name
    print '=' * 80
    print source
    print
```

19.2.5 包

要确定一个名字指示一个包还是常规的模块，可以使用 `is_package()`。

```
import zipimport
```

```
importer = zipimport.zipimporter('zipimport_example.zip')
for name in ['zipimport_is_package', 'example_package']:
    print name, importer.is_package(name)
```

在这里，`zipimport_is_package` 来自一个模块，`example_package` 是一个包。

```
$ python zipimport_is_package.py
```

```
zipimport_is_package False
example_package True
```

19.2.6 数据

有些情况下，发布源模块或包时还需要提供非代码的数据。图像、配置文件、默认数据和测试固件就是这样一些例子。通常，可以用模块的 `__path__` 或 `__file__` 属性来查找这些数据文件（相对于代码安装目录）。

例如，对于一个“正常的”模块，可以由导入的包的 `__file__` 属性构造文件系统路径，如下所示。

```
import os
import example_package

# Find the directory containing the imported
# package and build the data filename from it.
pkg_dir = os.path.dirname(example_package.__file__)
data_filename = os.path.join(pkg_dir, 'README.txt')

# Find the prefix of pkg_dir that represents
# the portion of the path that does not need
# to be displayed.
dir_prefix = os.path.abspath(os.path.dirname(__file__) or os.getcwd())
if data_filename.startswith(dir_prefix):
    display_filename = data_filename[len(dir_prefix)+1:]
else:
```

```
display_filename = data_filename
```

```
# Read the file and show its contents.
print display_filename, ':'
print open(data_filename, 'r').read()
```

输出取决于示例代码位于文件系统的哪个位置。

```
$ python zipimport_get_data_nozip.py
```

```
example_package/README.txt :
```

```
This file represents sample data which could be embedded in the ZIP
archive. You could include a configuration file, images, or any other
sort of noncode data.
```

如果 `example_package` 从 ZIP 归档导入而不是从文件系统加载, 则无法使用 `__file__`。

```
import sys
sys.path.insert(0, 'zipimport_example.zip')

import os
import example_package
print example_package.__file__
data_filename = os.path.join(os.path.dirname(example_package.__file__),
                             'README.txt')

print data_filename, ':'
print open(data_filename, 'rt').read()
```

包的 `__file__` 指向 ZIP 归档, 而不是一个目录, 所以构建指向 `README.txt` 文件的路径时, 会给出错误的值。

```
$ python zipimport_get_data_zip.py
```

```
zipimport_example.zip/example_package/__init__.pyc
zipimport_example.zip/example_package/README.txt :
Traceback (most recent call last):
  File "zipimport_get_data_zip.py", line 40, in <module>
    print open(data_filename, 'rt').read()
IOError: [Errno 20] Not a directory:
'zipimport_example.zip/example_package/README.txt'
```

要获取文件, 一种更可靠的方式是使用 `get_data()` 方法。可以通过导入的模块的 `__loader__` 属性访问加载模块的 `zipimporter` 实例。

```
import sys
sys.path.insert(0, 'zipimport_example.zip')

import os
import example_package
print example_package.__file__
print example_package.__loader__.get_data('example_package/README.txt')
```

`pkgutil.get_data()` 使用这个接口来访问包中的数据。

```
$ python zipimport_get_data.py
```

```
zipimport_example.zip/example_package/__init__.pyc
```

This file represents sample data which could be embedded in the ZIP archive. You could include a configuration file, images, or any other sort of noncode data.

对于并非通过 `zipimport` 导入的模块，不会设置 `__loader__`。

参见：

`zipimport` (<http://docs.python.org/lib/module-zipimport.html>) 这个模块的标准库文档。

`imp` (19.1 节) 其他与导入相关的函数。

PEP 302 (www.python.org/dev/peps/pep-0302) 新的导入 hook。

`pkgutil` (19.3 节) 提供 `get_data()` 的一个更通用的接口。

19.3 pkgutil——包工具

作用：增加到一个特定包的模块搜索路径，并处理包中包含的资源。

Python 版本：2.3 及以后版本

`pkgutil` 模块包含一些函数，可以改变 Python 包的导入规则，并从包中发布的文件加载非代码资源。

19.3.1 包导入路径

`extend_path()` 函数可以用来修改搜索路径，并改变从包导入子模块的方式，从而能结合多个不同的目录，使它们就好像是一个目录一样。利用这个函数，可以用包的开发版本覆盖已安装版本，或者将平台特定的模块与共享模块结合为单个包命名空间。

调用 `extend_path()` 最常用的方式是在包的 `__init__.py` 中添加下面两行代码。

```
import pkgutil
__path__ = pkgutil.extend_path(__path__, __name__)
```

`extend_path()` 会扫描 `sys.path` 来查找目录，其中要包括对应指定包（作为第二个参数）命名的子目录。这个目录列表与作为第一个参数传入的路径值结合，作为一个列表返回，很适合作为包导入路径。

名字为 `demopkg` 的示例包中包括以下文件：

```
$ find demopkg1 -name '*.py'
```

```
demopkg1/__init__.py
demopkg1/shared.py
```

`demopkg1` 中的 `__init__.py` 文件包含 `print` 语句，用来显示修改之前和之后的搜索路径，以强调二者的差别。


```

import pkgutil
import pprint

print 'demopkg1.__path__ before:'
pprint.pprint(__path__)
print

__path__ = pkgutil.extend_path(__path__, __name__)

print 'demopkg1.__path__ after:'
pprint.pprint(__path__)
print

```

extension 目录（增加了 demopkg 的特性）还包含另外 3 个源文件。

```
$ find extension -name '*.py'
```

```

extension/__init__.py
extension/demopkg1/__init__.py
extension/demopkg1/not_shared.py

```

下面这个简单的测试程序导入 demopkg1 包。

```

import demopkg1
print 'demopkg1', demopkg1.__file__

try:
    import demopkg1.shared
except Exception, err:
    print 'demopkg1.shared : Not found (%s)' % err
else:
    print 'demopkg1.shared', demopkg1.shared.__file__

try:
    import demopkg1.not_shared
except Exception, err:
    print 'demopkg1.not_shared: Not found (%s)' % err
else:
    print 'demopkg1.not_shared:', demopkg1.not_shared.__file__

```

直接从命令行运行这个测试程序时，找不到 not_shared 模块。

注意：这些例子中未给出完整的文件系统路径，而是做了缩写，来强调有变化的部分。

```
$ python pkgutil_extend_path.py
```

```

demopkg1.__path__ before:
['.../PyMOTW/pkgutil/demopkg1']

```

```
demopkg1.__path__ after:
```

```
['.../PyMOTW/pkgutil/demopkg1']
```

```
demopkg1      : .../PyMOTW/pkgutil/demopkg1/__init__.py
demopkg1.shared : .../PyMOTW/pkgutil/demopkg1/shared.py
demopkg1.not_shared: Not found (No module named not_shared)
```

不过，如果将 `extension` 目录添加到 `PYTHONPATH`，再次运行这个程序，会生成不同的结果。

```
$ export PYTHONPATH=extension
$ python pkgutil_extend_path.py
```

```
demopkg1.__path__ before:
['.../PyMOTW/pkgutil/demopkg1']
```

```
demopkg1.__path__ after:
['.../PyMOTW/pkgutil/demopkg1',
 '.../PyMOTW/pkgutil/extension/demopkg1']
```

```
demopkg1      : .../PyMOTW/pkgutil/demopkg1/__init__.pyc
demopkg1.shared : .../PyMOTW/pkgutil/demopkg1/shared.pyc
demopkg1.not_shared: .../PyMOTW/pkgutil/extension/demopkg1/not_
shared.py
```

`extension` 目录中的 `demopkg1` 版本已经添加到搜索路径，所以可以从中找到 `not_shared` 模块。

以这种方式扩展路径对于结合使用平台特定的包和公共包会很有用，特别是当平台特定的包版本中包含 C 扩展模块时。

19.3.2 包的开发版本

改进一个项目时，通常需要测试对已安装包的修改。将已安装版本替换为开发版本可能是个糟糕的想法，因为开发版本不一定正确，而且系统上的其他工具可能会依赖于已安装的包。

可以使用 `virtualenv` 在开发环境中配置一个完全独立的包副本，不过对于小的修改，建立这样一个包含所有依赖包的虚拟环境开销可能太大。

还有另一种选择，对于正处于开发状态的包，可以使用 `pkgutil` 修改其中模块的搜索路径。不过，在这种情况下，路径必须逆向设置，从而使开发版本覆盖已安装版本。

给定一个包 `demopkg2`，如下：

```
$ find demopkg2 -name '*.py'

demopkg2/__init__.py
demopkg2/overloaded.py
```

其中，正在开发的函数位于 `demopkg2/overloaded.py`，已安装版本包含：

```
def func():
    print 'This is the installed version of func().'
```

而 `demopkg2/__init__.py` 包含：

```
import pkgutil
```

```
__path__ = pkgutil.extend_path(__path__, __name__)
__path__.reverse()
```

`reverse()` 用来确保扫描默认位置之前，先扫描由 `pkgutil` 添加到搜索路径的目录来完成导入。这个程序会导入 `demopkg2.overloaded` 并调用 `func()`。

```
import demopkg2
print 'demopkg2', demopkg2.__file__

import demopkg2.overloaded
print 'demopkg2.overloaded:', demopkg2.overloaded.__file__
print
demopkg2.overloaded.func()
```

如果运行时没有做任何特殊的路径处理，会由 `func()` 的已安装版本生成输出。

```
$ python pkgutil_devel.py
```

```
demopkg2          : .../PyMOTW/pkgutil/demopkg2/__init__.py
demopkg2.overloaded: .../PyMOTW/pkgutil/demopkg2/overloaded.py
```

开发目录包含以下内容：

```
$ find develop -name '*.py'
```

```
develop/demopkg2/__init__.py
develop/demopkg2/overloaded.py
```

`overloaded` 的修改版本如下：

```
def func():
    print 'This is the development version of func().'
```

如果 `develop` 目录在搜索路径中，运行测试程序时则会加载这个版本。

```
$ export PYTHONPATH=develop
$ python pkgutil_devel.py
```

```
demopkg2          : .../PyMOTW/pkgutil/demopkg2/__init__.pyc
demopkg2.overloaded: .../PyMOTW/pkgutil/develop/demopkg2/overloaded.pyc
```

19.3.3 用 PKG 文件管理路径

第一个例子展示了如何使用 `PYTHONPATH` 中包含的额外目录扩展搜索路径。还可以使用包含目录名的 `*.pkg` 文件将目录添加到搜索路径。PKG 文件类似于 `site` 模块使用的 PTH 文件，其中可以包含要添加到包搜索路径的目录名，每行一个目录名。

对于第一个例子中应用的特定于平台的部分，建立这部分的结构还有一种方法：对于各个操作系统分别使用一个单独的目录，并包含一个 `.pkg` 文件来扩展搜索路径。

这个例子使用了同样的 `demopkg1` 文件，还包括以下文件。

```
$ find os_* -type f
```

```
os_one/demopkg1/__init__.py
os_one/demopkg1/not_shared.py
os_one/demopkg1.pkg
os_two/demopkg1/__init__.py
os_two/demopkg1/not_shared.py
os_two/demopkg1.pkg
```

PKG 文件命名为 `demopkg1.pkg`，使之与要扩展的包匹配。包名和 PKG 文件名中都包含：
`demopkg`

这个演示程序显示了所导入的模块的版本。

```
import demopkg1
print 'demopkg1:', demopkg1.__file__

import demopkg1.shared
print 'demopkg1.shared:', demopkg1.shared.__file__

import demopkg1.not_shared
print 'demopkg1.not_shared:', demopkg1.not_shared.__file__
```

可以用一个简单的包装器脚本在这两个包之间进行切换。

```
#!/bin/sh

export PYTHONPATH=os_${1}
echo "PYTHONPATH=$PYTHONPATH"
echo

python pkgutil_os_specific.py


指定参数 “one” 或 “two” 运行时，路径会调整。
$ ./with_os.sh one
```

```
PYTHONPATH=os_one

demopkg1.__path__ before:
['.../PyMOTW/pkgutil/demopkg1']

demopkg1.__path__ after:
['.../PyMOTW/pkgutil/demopkg1',
 '.../PyMOTW/pkgutil/os_one/demopkg1',
 'demopkg']

demopkg1          : .../PyMOTW/pkgutil/demopkg1/__init__.pyc
demopkg1.shared   : .../PyMOTW/pkgutil/demopkg1/shared.pyc
demopkg1.not_shared: .../PyMOTW/pkgutil/os_one/demopkg1/not_shared.pyc
```



```

$ ./with_os.sh two

PYTHONPATH=os_two

demopkg1.__path__ before:
['.../PyMOTW/pkgutil/demopkg1']

demopkg1.__path__ after:
['.../PyMOTW/pkgutil/demopkg1',
 '.../PyMOTW/pkgutil/os_two/demopkg1',
 'demopkg']

demopkg1          : .../PyMOTW/pkgutil/demopkg1/__init__.pyc
demopkg1.shared   : .../PyMOTW/pkgutil/demopkg1/shared.pyc
demopkg1.not_shared: .../PyMOTW/pkgutil/os_two/demopkg1/not_shared.pyc

```

PKG 文件可以出现在正常搜索路径的任何位置，所以还可以用当前工作目录中的一个 PKG 文件来包含一个开发树。

19.3.4 嵌套包

对于嵌套包，只需要修改顶级包的路径。例如，对于以下目录结构：

```

$ find nested -name '*.py'

nested/__init__.py
nested/second/__init__.py
nested/second/deep.py
nested/shallow.py

```

其中 `nested/__init__.py` 包含：

```

import pkgutil

__path__ = pkgutil.extend_path(__path__, __name__)
__path__.reverse()

```

开发树如下：

```

$ find develop/nested -name '*.py'

develop/nested/__init__.py
develop/nested/second/__init__.py
develop/nested/second/deep.py
develop/nested/shallow.py

```

`shallow` 和 `deep` 模块都包含一个函数来打印一条消息，指示消息来自已安装版本还是开发版本。

以下测试程序用来测试这些新包。

```

import nested

import nested.shallow

```

```

print 'nested.shallow:', nested.shallow.__file__
nested.shallow.func()

print
import nested.second.deep
print 'nested.second.deep:', nested.second.deep.__file__
nested.second.deep.func()

```

运行 `pkgutil_nested.py` 时，如果未对路径做任何处理，就会使用这两个函数的已安装版本。

```
$ python pkgutil_nested.py
```

```

nested.shallow: .../PyMOTW/pkgutil/nested/shallow.pyc
This func() comes from the installed version of nested.shallow

```

```

nested.second.deep: .../PyMOTW/pkgutil/nested/second/deep.pyc
This func() comes from the installed version of nested.second.deep

```

将 `develop` 目录添加到路径时，这两个函数的开发版本会覆盖已安装版本。

```

$ export PYTHONPATH=develop
$ python pkgutil_nested.py

```

```

nested.shallow: .../PyMOTW/pkgutil/develop/nested/shallow.pyc
This func() comes from the development version of nested.shallow

```

```

nested.second.deep: .../PyMOTW/pkgutil/develop/nested/second/deep.pyc
This func() comes from the development version of nested.second.deep

```

19.3.5 包数据

除了代码之外，Python 包还可以包含数据文件，如模板、默认配置文件、图像以及包中代码使用的其他支持文件。利用 `get_data()` 函数，可以采用一种与格式无关的方式访问文件中的数据，所以不论包作为一个 EGG 发布，还是作为一个冰冻二进制包的一部分，或者是作为文件系统上的常规文件，都没有任何影响。

对于一个包含 `templates` 目录的包 `pkgwithdata`，

```
$ find pkgwithdata -type f
```

```

pkgwithdata/__init__.py
pkgwithdata/templates/base.html

```

其中文件 `pkgwithdata/templates/base.html` 包含一个简单的 HTML 模板。

```

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>

```

```
<p>This is a sample data file.</p>
```

```
</body>
```

```
</html>
```

这个程序使用 `get_data()` 获取模板内容，并打印出来。

```
import pkgutil
```

```
template = pkgutil.get_data('pkgwithdata', 'templates/base.html')
```

```
print template.encode('utf-8')
```

`get_data()` 有两个参数，一个参数是包的加点名，另一个参数是相对于包顶级目录的文件名。返回值是一个字节序列，所以在打印之前会编码为 UTF-8。

```
$ python pkgutil_get_data.py
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
```

```
<html> <head>
```

```
<title>PyMOTW Template</title>
```

```
</head>
```

```
<body>
```

```
<h1>Example Template</h1>
```

```
<p>This is a sample data file.</p>
```

```
</body>
```

```
</html>
```

`get_data()` 与发布格式无关，因为它使用 PEP 302 中定义的导入 hook 来访问包内容。可以使用任何提供了导入 hook 的加载工具，包括 `zipfile` 中的 ZIP 归档导入工具。

```
import pkgutil
```

```
import zipfile
```

```
import sys
```

```
# Create a ZIP file with code from the current directory
# and the template using a name that does not appear on the
# local filesystem.
```

```
with zipfile.ZipFile('pkgwithdatainzip.zip', mode='w') as zf:
    zf.writepy('.')
    zf.write('pkgwithdata/templates/base.html',
            'pkgwithdata/templates/fromzip.html',
            )
```

```
# Add the ZIP file to the import path.
```

```
sys.path.insert(0, 'pkgwithdatainzip.zip')
```

```
# Import pkgwithdata to show that it comes from the ZIP archive.
```

```
import pkgwithdata
```

```
print 'Loading pkgwithdata from', pkgwithdata.__file__
```

```
# Print the template body
print '\nTemplate:'
data = pkgutil.get_data('pkgwithdata', 'templates/fromzip.html')
print data.encode('utf-8')
```

这个例子使用 `PyZipFile.writepy()` 创建一个 ZIP 归档，其中包含 `pkgwithdata` 包的一个副本，它包括模板文件的一个重命名的版本。在使用 `pkgutil` 加载模板并打印之前，将这个 ZIP 归档添加到导入路径。关于如何使用 `writepy()` 的更多细节，可以参考有关 `zipfile` 的讨论。

```
$ python pkgutil_get_data_zip.py
```

```
Loading pkgwithdata from pkgwithdatainzip.zip/pkgwithdata/__init__.pyc
```

```
Template:
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>

<p>This is a sample data file.</p>
</body>
</html>
```

参见:

`pkgutil` (<http://docs.python.org/lib/module-pkgutil.html>) 这个模块的标准库文档。
`virtualenv` (<http://pypi.python.org/pypi/virtualenv>) Ian Bicking 提供的虚拟环境脚本。
`distutils` Python 标准库的打包工具。
`Distribute` (<http://packages.python.org/distribute/>) 下一代打包工具。
`PEP 302` (www.python.org/dev/peps/pep-0302) 新的导入 hook。
`zipfile` (8.5 节) 创建可导入的 ZIP 归档。
`zipimport` (19.2 节) 这个导入工具可以导入 ZIP 归档中的包。

